Debian Library Packaging guide

April 13, 2007

Chapter 1

Introduction

This guide tries to illustrate and illuminate the problems related to library packaging to be clear to the Developers of Debian Project, to hopefully raise the general awareness, and to fill the gap of Debian documentation lacking in the direction of library package.

Hopefully this document will improve and become more accurate as criticisms come. The document will hopefully improve the general quality of Debian, and provide a good reading for Debian developers, instead of the "don't even dare packaging libraries if you are a newbie" policy, which used to be the air in debian-devel mailing list before this document was born back in 2002.

Latest version of this document is currently available at http://www.netfort.gr.jp/˜dancer/column/libpkg-guide/libpkg-guide.html <`http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.html`> (PDF version) <`http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.pdf`> (XML source) <`http://www.netfort.gr.jp/~dancer/column/libpkg-guide/libpkg-guide.xml`>

Chapter 2

Traits of Debian

First, let us check what Debian is from the perspective of maintaining library packages, and why Debian is different to other systems.

Debian is a binary-distribution-oriented system. The binaries are created from source packages and are the ones that get distributed. The binaries are from the source as a snapshot for one architecture and several copies of the binaries are built from the same source at different times for different architectures, sometimes even against different versions of libraries.

Thus, unless one is careful, one has no control over which version of specific development package a binary will be compiled with.

The SONAME of a library will help, as we will see in a moment.

From the point of a single distribution all packages should use the same version to reduce the number of installed packages and increase the possibility of sharing loaded libraries. And from the viewpoint of a distribution which can be upgraded, several different incompatible versions of shared libraries should be able to coexist.

Debian has a packaging scheme for libraries, with having libfooX package for run-time required library files, and libfooX-dev package for build-time required library files.

Other systems have a different approach to the problem. Some systems try to rebuild the whole system in one run, compiling against one version of single shared library. This is an ideal way, but requires a lot of resource to rebuild all packages in the archive.

Some other systems (including the ones which call their shared libraries .DLL) don't allow for upgrades, and assume that libraries are never upgraded, and even if they are upgraded they will be always compatible. Such systems are susceptible to random, very difficult to track, errors.

Chapter 3

Recommended reading before doing library packages

For detailed information on libraries, it is recommended to read the info page for libtool, which is contained in the libtool-doc package. It explains many things in detail, and talks about generic aspects of library programs not specific to Debian. Reading **ld** documentation in binutils-doc package is also interesting.

For Debian packaging backgrounds, please read the respective documentation for Debian Developers. Namely the Debian Policy, the New Maintainers' Guide, and the Developer's Reference. (An experience of having your package broken by some random library upgrade might be a plus, but hopefully this document will give you an idea of what kind of disaster happens when shared libraries break.)

Chapter 4

Contents of the shared library package and development package

In this chapter, what files are contained in which package is explained.

4.1 Files in lib* package

In the lib* (e.g. in this text, libfooX is used as an example, foo being the name of the package and X being a numeric number.) package, only the runtime library, and the files necessary to use the runtime library should be included in such a way that different versions of the runtime library can be co-installed on a user's system.

Usually, it contains the library file itself, somehow called `libfoo.so.X.X.X` and its symlink `libfoo.so.X` which matches the SONAME. [1]

When plugins and runtime binaries exist that are essential for using the shared library at runtime, they should be placed under a directory that can be derived uniquely from the SONAME. Usually that means they should reside under `/usr/lib/libfooX/` where libfooX is the full package name for the library package.

4.2 -DEV package

-DEV package (e.g. libfooX-dev) should contain the development symlink used when linking, static libraries, and header files, and if they exist, package configuration scripts.

Table 4.1 Annotated list of files that usually reside in -DEV package

files	meaning
usr/lib/*.so	development linkage file, used when other programs are linked with -lxxx
usr/lib/*.a	static link files
usr/lib/*.la	libtool linkage information file
usr/include/*	Development include files
usr/bin/*-config	Some configuration script used in obtaining the library paths, like `gtk-config`
usr/lib/pkgconfig/*.pc	Some information required for **pkgconfig**

[1]The SONAME may or may not end in a numerical value depending on the linkage option, and it is usually necessary to use **objdump** to really check it out.

4.3 Other files, plugins, runtime binaries

Usually upstream shared library packages contain some documentation and example runtime binaries. They should not reside in the runtime shared library package. They should be put in the -DEV package, or another package that does not have a SONAME version number appended on it, such as `libfoo-runtime`

This is because the ability of runtime shared library package to upgrade and coexist suffers if the binaries are included in the runtime shared library package.

There are plugin files, and binaries that are required by the shared library at run-time, which cannot be split out from the shared library. They are placed under a versioned directory inside `/usr/lib`. Examples include `/usr/lib/pango/1.0.0/modules/pango-arabic-ft2.so` for pango-1.0-0 modules, `/usr/lib/vdkbuilder2/libvdkbcalendar.so.2` for vdkbuilder2 modules, etc.

`/libexec` directory in GNU Coding Standards was designed to allow versioned binaries to exist. However, use of `libexec` is not currently allowed under current Debian policy. Use subdirectories of `/usr/lib/libfooX`.

4.4 Example of which packages the files belong to when using libtool

It is not immediately clear which files go to where when using libtool. The following table shows a list.

Table 4.2 The relationship between the link-time libtool command-line option and the actual file contents of each package

The libtool command-line	libfoo package name	libfoo package contents	libfoo-dev package name	libfoo-dev package contents
-export-dynamic -version-info 0:0:0 -release 1.2.3	libfoo-1.2.3-0	libfoo-1.2.3.so.0, libfoo-1.2.3.so.0.0.0	libfoo-1.2.3-0-dev	libfoo.so, libfoo.a, libfoo.la
-export-dynamic -release 1.2.3	libfoo-1.2.3	libfoo-1.2.3.so	libfoo-1.2.3-dev	libfoo.so, libfoo.a, libfoo.la
-export-dynamic -version-info 0:0:0	libfoo0	libfoo.so.0, libfoo.so.0.0.0	libfoo0-dev	libfoo.so, libfoo.a, libfoo.la

Chapter 5

shared library packages

5.1 SONAMEs, API and ABI

In most cases, if a package version matches the SONAME, it is a sign that there is a problem with the versioning scheme. Scrap it, and bash the upstream with the libtool manual. It is usually a good sign that either he has not read the manual thoroughly, or he has not understood it, or both.

A SONAME is an information stored in a shared library, which can be seen with the command **objdump -p** *`filename`* | **grep SONAME**. The value is referenced from binaries and shared libraries when linking, and embedded in the NEEDED fields, which can be seen with **objdump -p** *`filename`* | **grep NEEDED**. A soname usually looks something like `libfoo.so.0`.

If a package keeps the same SONAME, it should mean that the *BINARY COMPATIBILITY IS KEPT* (however, it's not always the case).

If a new version of a library package breaks a currently existing and working package (the ABI), the SONAME version number should be bumped up, or the change be reverted, or both. By bumping up the SONAME version number, the old binaries which used to link to the old version of the library should be able to run with the old library, and the new and the old libraries can coexist.

Signs of binary incompatibility include: function declaration change, change of "struct" contents, and changing semantics of functions (hard to detect).

If it only requires a source rebuild, it is called a ABI breakage. When even a rebuild is not enough, and there is a source-level change required for applications to work with the new version of the library, it is called a API breakage, and a different -DEV package name should be chosen for the new version of the shared library package.

5.2 Choosing which method to use for versioning

The upstream authors have the liberty of choosing two major methods for versioning using libtool. -version-info, and -release. -release is used for unstable libraries that change ABI on every new release. However, such unstable library package usually don't belong in Debian, because it will require a rebuild in every dependent package against the new library package.

-release is recently used more for signifying major releases. Due to the serial nature of -version-info, SONAME version numbers usually get quite large fairly quickly. Using new -release value in major library release, the SONAME version numbers can be re-set to zero. Some people prefer the lower numbers.

5.3 Naming shared library packages

The policy documents how to name library packages. `"lib[libraryname][SONAME-version-number]"` like `"libc6"` for `/lib/libc.so.6`

However, there are packages which contain libraries that look like this:

```
/usr/lib/libfoo-1.2.so.0
```

This is a library with a name of `libfoo-1.2`, and a SONAME version number of "0". The current practice in packaging such package is to have `libfoo-1.2-0` or `libfoo1.2-0` as the package name, and the recommended practice is to have `libfoo-1.2-0`

Do not make it `libfoo1.20`, since it is ambiguous.

The package name should match the shared library SONAME. It can be deduced with **sed s/\.so\./-/** and removing "-" later when removing it will not result in consecutive numbers. Or by using the following code snippet from Steve Langasek.

```
$ objdump -p /path/to/libfoo-bar.so.1.2.3 | \
  sed -n -e's/^[[:space:]]*SONAME[[:space:]]*//p' | \
  sed -e's/\([0-9]\)\.so\./\1-/; s/\.so\.//'
```

Table 5.1 Example match between SONAME and package name

SONAME	package name
libfoo-1.2.3.so.4	libfoo-1.2.3-4
libfoo-1.2.3.so	libfoo-1.2.3
libfoo.so.4	libfoo4
libfoo.so	libfoo (But please don't introduce such package!)

There are packages like libc6, which contain multiple shared libraries in one package. This is not encouraged. [1] It becomes more complex and more difficult to handle complex upgrade patterns. bug#141275, omniorb package contained several different libraries with different SONAME version numbering policies. <http://bugs.debian.org/142175>

There has been a history of packages which were named with the source package name, but it is better practice to name the package according to the library name, for consistency. Some old packages are not named this way, such as aalib-dev, but new packages should follow the scheme of using the library SONAME.

For an example of a package which migrated from single-package library file, see xlibs. xlibs in Debian 3.0 was one package containing many shared libraries, and it is split into multiple packages in later releases of Debian. The xlibs package itself is kept as a package which depends on the library packages, so that compatibility and smooth upgrade are ensured.

5.4 Dependency of shared libraries, and indirect dependencies.

Shared libraries should depend on what shared libraries it depends upon. There are indirect dependencies, where libA depends on libB and libB depends on libC. In that case, a dependency of libA on libC should not be necessary, since the dynamic loader will take care of processing such dependencies at run-time. From the Debian perspective, having the indirect dependency turned into an explicitly dependency adds to the maintenance overhead, since if library libC is upgraded to become libC2, and libB is updated to work with libC2, libA will be linked with libC and libC2 simultaneously. This usually causes a problem

[1] This is the case unless it is confident that shared libraries will not change interfaces independently, or compatibility issues are carefully handled. In general, when shared libraries are split, there is no reason upstream will keep changes to interfaces synchronised.

due to having multiple instances of similar functions into the memory, and is wasteful having to load an unused bit of code. It will require a rebuild of libA to fix the situation, which means Debian archive will need quite a few binNMUs.

There are a few tricks to avoid this problem. One is using –as-needed option, and another is using pkgconfig scripts properly.

5.5 Shared libraries should link with shared libraries it depends on

Sometimes upstream decides that it is a good idea not to link shared libraries with the depending shared libraries, which was the case for libpng[2] , and some BSD variants. That breaks a few things, and thus is not desirable. The following are examples of what breaks.

5.5.1 dlopen

dlopen loads shared library, and allows resolving functions in the shared library at run time. The shared library dependency information is used to resolve the dependencies. Not linking the required shared libraries with the shared library will result in missing symbols. This bug does not appear very much with “RTDL_GLOBAL”, but appears more with loading with “RTDL_LOCAL”.

5.5.2 symbol versioning

Symbol versioning works at shared library link time, so if the shared library is not linked with the symbol-versioned shared libraries, symbol versioning will not work.

5.5.3 linking with -Bsymbolic

-Bsymbolic tries to link things with local namespace, and resolves function symbols within the shared library. Not linking will break it. [3]

5.6 What to put in shlibs file

The policy section on ”Shared Libraries,” and ”Handling shared library dependencies - the ‘shlibs’ system” has explanations about shlibs file. **dh_makeshlibs** creates the required shlibs file, in a simple case.

In shlibs file, put something like

```
libpcap 0 libpcap0 (>= 0.6.1-1)
```

When compatibility breaks, change the SONAME and make it look like

```
libpcap 1 libpcap1
```

or, at least give it a different Debian package name

[2]libpng maintainers considered in 1.2.5 that linking libpng with -lz was absurd, and removed it. It made a lot of packages fail to build. 166489 <http://bugs.debian.org/166489>

[3]This is generally a bad idea and tends to break due to its way of resolving symbols. Use of symbol versioning is recommended.

```
libpcap 0 libpcap0a
```

and create a new libpcap0a package, conflicting with libpcap0

Doing something like :

```
libpcap 0 libpcap0 (>= 0.6.1-1), libpcap0 (<< 0.7.0)
```

is not good because it will not survive the epoch in the version, and is really unnecessary if SONAMEs are used properly.

This is discussed in binary compatibility in detail.

Chapter 6

Development (-DEV) packages

6.1 -DEV package names

Package maintainer has two options when naming a shared library -DEV package. One is to name the package after the library name and not include the full SONAME, like: libfoo-dev.

When naming the package after the full SONAME version numbers in the package name, the name is constructed by adding the '-dev' suffix to the library package name. Like the following:

```
   Package: libfoo2-dev
   Provides: libfoo-dev
   Conflicts: libfoo-dev
```

The latter is preferred if the library package is widely used, and the API is subject to change.

Each -DEV package should conflict and provide `libfoo` so that no two -DEV package can coexist. This is because the `.so` symlink and other files use the same name on shared library package of different SONAME. This is unless the package takes enough care to not have duplicate filenames, including `.so` symlink, and header file paths. Gnome libraries give a good example of such a setup.

An example command-line to generate the -DEV package name from a shared library SONAME is as follows:

```
$ objdump -p /usr/lib/libshared.so | \
  sed -n 's/^[[:space:]]*SONAME[[:space:]]*//p' | \
  sed 's/\(0-9\)\.so\./\1/; s/\.so\.//; s/$/-dev/'
libshared0-dev
```

6.2 -DEV package dependencies

The -DEV package would usually declare Depends: relationship on all -DEV packages for libraries that the library package directly depends upon, with the specific SONAME version that the library package is linked against. This includes libc-dev. [1]

The dependency is required to make things such as statically linked libraries to work, and C header file inclusions. Libtool .la files require dependencies to be present also. [2] [3]

[1] A package should depend on libc-dev, without versioned depends, or generate different dependencies depending on architectures. Not all architectures have libc-dev as libc6-dev.

[2] Shared library .so files do not actually require the depending -dev packages at link time, so if upstream was careful enough, it is possible to remove the dependency, if support for static linking is to be dropped.

[3] libtool .la require all recursive dependencies. pkgconfig .pc files however, do not. See pkgconfig section for details

e.g.: libfoo2-dev -> libbar2-dev because libfoo2 depends on libbar2

This dependency helped in the case of libpng2 and libpng3 to avoid libpng2-dev and libpng3-dev to be installed at the same time, so that problematic packages could be detected easily. [4]

When `libfoo2-dev` that can compile with libbar3-dev is required, the SONAME version number of libfoo should be bumped up, or a new package containing libfoo linked with `libbar3-dev` conflicting with the original `libfoo2` needs to be created. However this is not enough, as discussed in the binary compatibility section.

`libfoo2-bar3-dev` (which is a development package for a shared library package `libfoo2-libbar3` which contains `libfoo.so.2` linked with `libbar.so.3`) which depends on `libbar3-dev` and has `libfoo2` and `libfoo2-bar3` conflicting with each other.

or

discuss with upstream to get: `libfoo3-dev` for depending upon `libbar3-dev`. Which is a better solution, which keeps cross-distribution binary-compatibility.

There is one exception for this section; the case of libraries with versioned symbols. For example, if a -DEV package depends on libdb2-dev, and libdb2-dev has symbols that are versioned to allow coexisting with libdb3, it may depend on libdb-dev, and not libdb2-dev.

6.3 Packages which Build-depend on a -DEV package

It is advised to

```
Build-Depends: libfoo[SONAME-version-number]-dev
```

(Which needs to be updated every time a new -DEV comes out, and the new SONAME becomes the standard, and the old one becomes obsolete)

or

```
   Build-Depends: libfoo[SONAME-version-number]-dev | libfoo-dev
```

(this can cause undesired effect of linking with a source-incompatible (API-incompatible) library version, i.e. a serious "cannot build from source" bug)

6.4 "Build-Depends: libfoo-dev" is not optimal

The question is: Are you really sure all past/present/future version of that -DEV package can be used with the source to build the package? (i.e. is your API so fixed in stone that it will never have to be changed?) If this is not true, some trouble will happen every time the -DEV package changes. [5]

[4] Bugs like Bug 146079 <http://bugs.debian.org/146079> go undetected when -DEV packages do not properly depend on other -DEV packages.

[5] Stephen Frost commented that: Build-Depend's should be *accurate* in that they map to the *API* that's required. Sometimes this works out as being the same as the SONAME, but that's not always the case. Multiple ABI's can be associated with a single API. This is actually the case with OpenLDAP which claims, at least, to have not broken backwards compatibility with the API since the 1.x days, though the ABI has changed a number of times. Of course, if a package depends on parts of the API that were added later they should version their build-depends appropriately. In general it'd probably actually be good to get away from having version numbers in -DEV package names based on the expectation that upstream will be similar to the OpenLDAP case, and in the event that the API is changed in a way which is not backwards compatible the library name may be changed in some other way.

6.5 -rpath considered harmful

Use of -rpath is usually discouraged in Debian, since having -rpath of /usr/lib will make the dynamic loader behave differently, and will have trouble working fine in cases of libc5-libc6 transition, and cases where multiple shared library versions exist under subdirectories of /usr/lib and are selected by ld.so under some criteria. It will potentially break behavior with multiarch, where shared libraries are not found in /usr/lib, but under /usr/lib64, /usr/lib32, or other places.

To remove -rpath in the upstream level, it is usually non-invasive to request upstream to special-case /usr/lib, to not add -rpath.

Richard Atterer summarized his points on his post to debian-devel. <http://lists.debian.org/debian-devel/2002/07/msg02030.html>

6.6 pkgconfig files

-DEV packages may contain pkgconfig files. pkgconfig is a tool to replace libfoo-config scripts, in a way which is integrated with autoconf.

pkgconfig has constructs for private shared library linkage, as opposed to libtool. libtool will require depending on all shared library packages recursively. Not depending on unneeded shared library is a plus for release management.

pkgconfig is preferred to libtool, and .la files are in the process of being phased out in favor of .pc files.

Removing .la files and replacing them by pkgconfig files remove the requirement. However, the transition needs to be coordinated in leaf-first order, or will cause problems found in libXcursor and libXrender. Bug# 363239 <http://bugs.debian.org/363239> and Bug# 363057 <http://bugs.debian.org/363057>.

6.7 Can I provide static link library only?

There are cases where the upstream provides only the static link libraries. However, doing so is not ideal, because it will result in binaries that cannot be rebuilt from source. If a newer static library is released since the last time a binary package was linked against it, the binary package contains code that can no longer be rebuilt from Debian source.

There are several reasons for providing static libraries, but it is best to avoid it, if it is technically possible. Unstable ABI is one reason to provide static libraries for, but please reconsider putting such a library package into a stable Debian distribution.

Providing -fPIC versions of static libraries for linking with shared libraries is a bad sign, because the "unstable interface" is now exported through another library's stable library interface.

zlib vulnerability (DSA122-1) required many packages to be rebuilt from source, because many packages were linked to it statically. It takes much resource to fix such bugs, and if it were linked dynamically, only one binary package had to be updated. <http://www.debian.org/security/>

Chapter 7

Handling upstream changes to shared libraries

In this chapter, methods of handling upstream shared libraries are discussed.

7.1 How to fix upstream packages with somewhat broken SONAMEs

Refer to libssl and other packages which used to handle it. They basically had SONAME version numbers which matched the package version, and every version: e.g. 0.9.4 and 0.9.6 had incompatibility. The solution was to create a SONAME containing 0.9.6, so that :

```
$ objdump -p /usr/lib/libssl.so.0.9.6 | grep SONAME
SONAME     libssl.so.0.9.6
```

libssl-dev contains the symlink `/usr/lib/libssl.so -> /usr/lib/libssl.so.0.9.6`

This way, binary programs linked with libssl.so via ”-lssl” command line option passed to gcc will be looking for libssl.so.0.9.6 at runtime.

It is quite important that Debian does not lose binary compatibility with other distributions, so changing the SONAME specifically for Debian is generally a bad idea. Discuss and convince the upstream to use a saner method for determining the SONAMEs.

It is however sometimes necessary to add a SONAME for the time-being until you have convinced the upstream. Make sure you choose a SONAME that is obviously Debian-specific, and be prepared for the eventual transition to the upstream-chosen SONAME.

7.2 When binary compatibility breaks

SONAME needs to be updated when the binary compatibility is broken.

When the library itself changes the interface, the SONAME needs to be changed, because the binary compatibility has changed.

Also, when the library that the library depends on has changed incompatibly, it means that the library depending on the changed library has changed incompatibly. i.e. if the library will need to link to a shared library with a different soname than it had previously, the soname needs to be modified.

The SONAME needs to be modified to reflect this change.

However, it is not always possible to increase the SONAME version number, possibly to remedy past problems, and experiences. To do that, it is also possible to change just the package name. Note that it is the best to modify the SONAME in the upstream level, because recompiling with a new package name will solve problems within the Debian packages, but it will not solve problems with the user compiled binaries in places such as `/usr/local/`. Also it might cause problems with software that is distributed in binary-only form, which expects to have some shared library with a specific interface. Debian supports the use of such binaries, and packages should not break them. Therefore the method described here should only be used as a last resort. It is better than changing the binary interface and not changing the library package name or soname [1]

It is strongly discouraged to create a shlibs file containing a (= VERSION) relationship, or (> VERSION) and (< VERSION) relationship. Such packages should not be released as stable.

`libfooX` may have been broken, and to fix it, introduce a new package `libfooXSOMETHING`. Alter the shlibs files so that building with `libfooX-dev` will cause the binary package to depend on `libfooXSOMETHING`, like the following:

```
libfoo X libfooXSOMETHING
```

Also `libfooXSOMETHING` should have the following package information:

```
Package: libfooXSOMETHING
Provides: libfooX
Conflicts: libfooX
```

to reflect that `libfooXSOMETHING` is not installable alongside with libfooX. A package rename is necessary, because such relationship is very difficult to express without a package rename, and without one, problems such as described in bug #155938 may occur.

And start recompiling every package that is linked against libfooX against the libfooX-dev, updating the Build-Depends accordingly (to build-depend on a version greater than the newly created libfooX-dev).

```
apt-cache rdepends libfooX
```

or

```
grep-available -F Depends -s Package,Depends libfooX
```

will give a rough idea of what needs to be done (although possibly not complete).

This whole process needs a lot of interaction between developers. The individual maintainers need to be notified, and some discussion and coordination through debian-release@lists.debian.org, and possibly debian-devel@lists.debian.org mailing list is recommended. It usually takes order of several months to get something like this fixed, and it is best to avoid such change.

Also note the implication on the “testing” release model of Debian. Library binary-package name changes currently require manual intervention and “hinting” for migration into the “testing” distribution, since a set of packages needs to migrate from “unstable” to “testing” in one set. This is true when the library source package name is unchanged, and the version in unstable cannot coexist with the version in “testing” in the source archive; meaning that the old package with the old soname will be removed from “testing”. This is due to the fact that multiple packages depend on a shared library, and these packages need to be updated at the same time.

[1] libsdl-image1.2 recompiled with libpng3 while it was previously linked with libpng2 and broke many packages

7.3 What kind of change is permitted without soname change and when do I need to change it?

This part of shared library packaging guide is focused more on shared library upstream maintainers, rather than Debian. Since Debian is one of the largest binary distribution around, Debian is the one who will experience problems from what others do.

There are cases where the SONAME does not have to change when the source code changed. For example, when a new function symbol is introduced and existing symbols are not modified, it is a backward-compatible change. Old programs linked to the library will work with the new library. New programs compiled against the new library will not work with the old library, so this needs to be noted in the shared library dependency. [2] This is the case where one would use dh_makeshlibs -V option; adding a (>= VERSION) relationship. [3]

Changes and effects is a rough list of cases.

7.4 What to do when SONAMEs change too often

There are some cases where the library SONAMEs change too often. It might be a legitimate thing, but the upstream may be doing it just for the sake of it. Check their modifications, and suggest to increase the SONAME version number only when the library has an incompatible change.

When you only see ChangeLog file and configure.in and such files being modified, and yet you see SONAME changes, it is a good sign that the upstream is not taking SONAMEs seriously.

There are libraries which are under heavy development. It is a pain anyway, because people have to follow it, accept that it is a pain. It is almost impossible to package a moving target into a stable distribution.

As a temporary measure such fast moving libraries can be built as .a libraries and statically linked to. This ensures that binaries contain the object files they were compiled against. Be careful though, while this removes the need of an ever increasing SONAME version number, doing this can cause problems later if these static libraries are used in shared objects of other packages. And also, this does not solve everything. Library packages are constantly evolving for a reason.

Using statically linked libraries open up a can of worms. Even if upstream does not come up with a shared library, it might be better to use the -release flag to add a Debian specific version string, like debian.20020512. Constructing the version number including the string debian, and the date should make the version number unique.

There was a problem with libgal SONAME changing too rapidly, which caused people to work around it in all sorts of strange ways. <http://lists.debian.org/debian-devel/2002/debian-devel-200201/msg01772.html>

[2] However, remember to change the last digit of the shared library file, since some Operating Systems other than Linux do not allow online replacement of files with the same name. One of the reasons for the libfoo.so.X symbolic link pointing to libfoo.so.X.Y.Z.

[3] librote (A thread discussing an example of compatible change.) <http://lists.debian.org/debian-devel/2005/06/msg02017.html>

Table 7.1 References for broken SONAME upgrades and packages

SDL	<http://lists.debian.org/debian-devel/2001/debian-devel-200110/msg00353.html>
libpng	<http://lists.debian.org/debian-devel/2002/debian-devel-200201/msg00243.html>, discussion about png transition and qt2. Bug 147707: <http://bugs.debian.org/147707> should libpng2 conflict with old gimp?, commenting about problems with sonames not changing when binary compatibility is broken. Bug 153813 <http://bugs.debian.org/153813>: libsdl-image1.2 relinked from libpng2 to libpng3 without changing the soname. causing libsdl-perl to break, causing frozen-bubble to break. Upgrade from woody will break even if this ad-hoc fix is installed. Bug 155938 <http://bugs.debian.org/155938>: libsdl-perl and libsdl-image1.2 in sarge (testing) got out of sync, although libsdl-image1.2 problem (linking with libpng3) was addressed with random recompilation of packages within sid, their migration to testing caused similar problem, because there was no dependency tracking information.
slang-utf8	Trying to fix slang post in debian-devel mailing list <http://lists.debian.org/debian-devel/2002/debian-devel-200201/msg02539.html>
libsmpeg0	libsmpeg0 naming problem, should have been libsmpeg0-4, bug 140572 <http://bugs.debian.org/140572>
liby2	liby2 contained liby2.so.7, or liby2.so.12, but the package name was same from potato to woody. http://bugs.debian.org/140753 (xship-wars, a package depending on liby2) <http://bugs.debian.org/140753>http://bugs.debian.org/138815 (liby2 fix) <http://bugs.debian.org/138815>
clanlib	clanlib SONAME version number is upgraded from .1 to .2 without changing package name: Bug 140976 <http://bugs.debian.org/140976>
libsnmpkit1	pconf-detect was broken by libsnmpkit1 when the package was silently upgraded with libsnmpkit.so.2 included. Obviously, the maintainer didn’t test the binaries, only compiled the shared libraries and uploaded. <http://bugs.debian.org/145462>
libmotif	libmotif: soname changed without package name change. <http://bugs.debian.org/150635> libmotif used to contain libXm.so.2, but it now contains libXm.so.3, with the same package name.
libvorbis	libvorbis0 package was split into libvorbisfile3 etc packages without changing package name. Causing many breakages. broken ogg123. <http://bugs.debian.org/154699>broken defendguin. <http://bugs.debian.org/154704>frozen-bubble breakage.. <http://bugs.debian.org/154744>libvorbis upgrade breaks many apps. <http://bugs.debian.org/154699>libsdl-mixer1.2 breakage. <http://bugs.debian.org/154680>xmms breakage. <http://bugs.debian.org/154765>

Table 7.2 Changes and effects

Change	SONAME	shared library filename	Debian versioning
Removing a function, changing a semantic of function	change	change	change
Changing a struct incompatibly	change	change	change
Adding a function	keep	change	Add depends on >=
Depending library changes SONAME (without versioned symbols)	change	change	change
Depending library changes SONAME (with versioned symbols)	keep	change	Add depends on >=
Changing byte-packing behavior, compiler options	change	change	change

Chapter 8

Consideration when building binary package and library package from single source

When building packages which have binaries linked against the shared library built from the same source, a trick is required to properly set the Depends: line.

Create a `debian/shlibs.local` file containing the necessary dependency information, and add the shared library location to LD_LIBRARY_PATH. `shlibs.local` should contain something in the line of SONAME-before.so SONAME-after.so package-name, which is documented in the policy manual.

When using debhelper, such can be achieved by: **dh_shlibdeps -LlibfooX -l${PWD}/debian/libfooX/usr/lib**

Chapter 9

Advanced linker tricks

9.1 Only linking shared library as needed

GNU ld has an extention to only link shared libraries that have used symbols, thus removing unnecessary linkages. Unneeded linkages are maintenance burden, and although it should really be fixed manually, having an automated help is usually a good idea. It can be used as the following:

```
LDFLAGS="-Wl,--as-needed" ./configure --prefix=/usr [...]
```

Note that currently libtool has a bug that it reorders argument, which breaks passing –as-needed option to ld. <http://bugs.debian.org/347650>

9.2 Only exporting required functions

It is possible to use -export-symbols-regex option of libtool to restrict symbols exported from the shared library. It is usually a good idea to do so since unexpected exported symbols can be a problem due to namespace and other issues. You can use this method, or the method explained in symbol versioning. To use this feature, it's simple. This is an example Makefile.am snippet for libSDL_mixer, to export only functions that match the regular expression (regex) **Mix_.***

```
libSDL_mixer_la_LDFLAGS =       \
[...]
        -export-symbols-regex Mix_.*
```

9.3 Symbol Versioning for shared libraries

Sometimes, multiple versions of shared library in distribution is a social problem. The reasoning being that other maintainers are not willing to rebuild their packages against new versions of shared libraries. However, there are cases where it is necessary to retain compatibility with older versions of shared library; especially when dlopen is used, and the application is not going to be restarted through upgrades. PAM, and some daemons are affected by this. This effectively requires most of Debian base system to be eventually versioned, to allow seamless upgrades.

A presentation was given by Steve Langasek in Debconf4, Brazil. <http://www.debconf.org/debconf4/talks/dependency-hell/index.html> The libpkg-guide originally presented a strategy of -DEV packages conflicting with each other; it did not scale very well, and caused much disruptions. To ease transition of changing library SONAMEs, it is possible to use versioned symbols in shared library and allow multiple versions of the same shared library to coexist within single application instance.

9.3.1 Symbol versioning

Symbol versioning is a linker trick where functions are treated as if they were prefixed by a specific symbol. For example, if there are two different libraries libA and libB providing one "initialize" function, it is possible to symbol-version them so that binaries will try to resolve libA@initialize and libB@initialize so that they can coexist within the same shared library namespace.

9.3.2 How to make a shared library package with versioned symbol

Adding options *-Wl,--version-script,dsh.ver* to gcc or libtool allows using the version script for versioning of shared library.

A version script will look like this:

```
HIDDEN {
local:
    __*;
    _rest*;
    _save*;
};

libdshconfig0 {
    *;
};
```

HIDDEN part is required to hide some symbols from being versioned. This example will add a version "libdshconfig0" to the exported symbols defined within the library. To verify that the symbols are versioned, **objdump -T** can be used. A snippet of output looks like this:

```
00000000      DO *UND*  00000004  GLIBC_2.0    stderr
00000e04 g    DF .text  00000048  libdshconfig0 free_dshconfig
00000dc4 g    DF .text  0000003f  libdshconfig0 free_dshconfig_internal
00000000 g    DO *ABS*  00000000  HIDDEN       HIDDEN
00000ed0 g    DF .fini  00000000  libdshconfig0 _fini
```

The name of the version symbol needs to be unique across all shared libraries within Debian, and it should be a good idea to derive the string from the shared library SONAME.

Not all Operating systems support versioned symbols, and usually making the script optional would be required. You may use autoconf/automake checks to make the option configurable. An example of doing such check can be found in libdshconfig. The following snippet is added to configure.ac to check for *--with-versioned-symbol* option to configure script invocation:

```
    vsymldflags=
    AC_MSG_CHECKING([version script options])
    AC_ARG_WITH([versioned-symbol],AC_HELP_STRING([--with-versioned-symbol],[Use versioned symbols]),[
    vsymldflags="-Wl,--version-script,dsh.ver -Wl,-O1"
    ])
    AC_SUBST(vsymldflags)
    AC_MSG_RESULT([${vsymldflags}])
```

The following snippet is added to Makefile.am to actually add ld flags for versioned symbol.

```
lib_LTLIBRARIES = libdshconfig.la
libdshconfig_la_LDFLAGS = -export-dynamic -version-info $(DSHCONFIG_SONAME) @vsymldflags@
```

9.3.3 Migration strategy

To enable symbol versioning on two different shared libraries which were previously not versioned, all binaries need to be rebuilt against the versioned shared library.

Enabling versioned symbol does not break binary compatibility, and it is be easier and less disruptive to just enable versioned symbols and rebuild all other binaries depending on the shared library, to avoid too much disruptions to the Debian archive. However, make sure enough time is given for the library to build to get all architectures synchronized. [1]

Note that, to be pedantic, the new library needs to conflict with binaries which were linked against the unversioned shared library. This is discussed in binary compatibility section. Say the library package was originally named libunversion0. The most pedantic way to ensure this effect is to change the shared library package name to libunversion0v, and conflict with libunversion0, so that every binary package built against the versioned shared library will show up in the Depends: field. This process will be disruptive and make many packages uninstallable, and should be used with care.

9.3.4 Possible problem cases

Symbol versioning only versions function/variable name symbols, and does not version structures etc, so it does not solve problems with mixed protocols, data structures and other things with mixed versions of shared library. Thus, it is not a silver bullet for all cases.

Symbol versioning only handles shared libraries, and it will not help with statically linked libraries.

Also there is a possibility of breakage in the transition phase. When binary A is linked against unversioned libpng.so.2 and libB, and old libB was linked against unversioned libpng.so.2. libB is now linked against versioned libpng.so.3, and upgrading libB may cause failures.

The dynamic linker is intelligent enough to handle cases when the new library is only available as versioned symbols, and the older library is only available as unversioned symbol.

Bug 140490 is for some symbols that should not be versioned. You need to be restrictive about what symbols to version. <http://bugs.debian.org/140490>

libdb transition was painful, but it was completed.

9.3.5 Supported architectures

All architectures that are in the Debian distribution support symbol versioning. But it should be noted that each architecture has its list of symbols that should not be versioned.

9.3.6 Some references on symbol versioning

”ELF symbol versioning with glibc 2.1 and later” from Ulrich Drepper <http://lists.debian.org/lsb-spec/1999/lsb-spec-199912/msg00017.html> and Info manual for ”ld” has some documentation.

[1]Note also that this assumes that user is going to perform dist-upgrade to upgrade all packages at once, and does not support the use of individual upgrades.

Chapter 10

dlopened modules and implication of dlopening

Many basic library packages freely use dlopen as a method for loading shared library symbols dynamically. Examples are glibc and PAM. There are several advantages and disadvantages. For advantages:

- Dynamic loading allows dynamic library dependency
- No fork/exec/pipe/ipc overhead for extra functionality, and allows easy code sharing
- Allows use of library code without intermediate interfacing

Disadvantages:

- Cannot take advantage of prelinking
- Introduces untested combinations of shared libraries
- Shared library being dlopened may change while the applications are running (e.g. when upgrading); glibc worked around this problem by giving an option of reloading affected binaries.

There were a few problem cases with this situation. PAM_ldap dlopened ldap libraries, which depended on a chain of shared libraries such as libssl. However, ssh and ldap did not agree on which version of libssl to depend upon, and ssh ended up loading two different libssl versions into the same function namespace when PAM was configured with ldap.

A problem happened when glibc changed incompatibly regarding libnss. An application that started up with the older version of glibc would crash, if upgraded version of libnss was loaded. libnss loading was triggered by nameserver lookups, for example.

Similar problem appeared when GTK themes used libpng2. Because GTK pixmap theme uses libpng2, and GTK applications dlopen theme engines, GTK applications that are linked with libpng3 resulted in load failures when the pixmap engine was specified as the theming engine.

This situation cannot be avoided with strict -DEV versioning and dependency as described in this paper.

If a library or application does a dlopen to use a module, that module and its chain of dependencies have a chance of two different versions of the same module being loaded at the same time. Unless RTDL_GROUP option for dlopen gets implemented and gets used, there are basically two options to avoid problems:

- Uniquely named symbols, differently between different SONAMES.
- Version the symbols using symbol versioning as described in Section 9.3.

Chapter 11

How shared library is loaded

This is an informational section. This section is targeted at Debian Developers who need to explain convincingly to upstream developers on why symbols should be versioned, and why things should be as it is. This section explains how an ELF shared library is loaded with Linux kernel and glibc. Ulrich Drepper's writeup on DSO's has a more detailed account on this topic. <http://people.redhat.com/drepper/dsohowto.pdf>

When an ELF header is found by ELF handler inside Linux Kernel [1] , it will invoke the executable found in the ELF .interp section. [2] The dynamic loader, which usually is /lib/ld.so.1; which itself usually is a static ELF binary. [3] Dynamic loader will interpret the ELF header and perform the rest of loading.

The functions from the shared library are loaded to memory in a relocated address. Which is determined at the time ld.so loads the shared library. [4] The important point to note is that the assembly command to call the subroutine will require a memory address where the code is located, but that is not calculated at the time of compiling, linking, or package building. ld.so uses the function names embedded in the shared library (called symbols) and resolves the symbols through name-matching.

Let us see with live example. The following is a binary `testprint` which is linked to the shared library `libshared` which defines some symbols like `shared_new`. The code itself looks like this when disassembled through gdb: [5]

```
Dump of assembler code for function main:
0x10001650 <main+0>:    stwu    r1,-32(r1)
0x10001654 <main+4>:    mflr    r0
0x10001658 <main+8>:    stw     r31,28(r1)
0x1000165c <main+12>:   stw     r0,36(r1)
0x10001660 <main+16>:   bl      0x10011c00 <shared_version>
0x10001664 <main+20>:   bl      0x10011be8 <shared_new>
0x10001668 <main+24>:   cmpwi   r3,0
0x1000166c <main+28>:   mr      r31,r3
0x10001670 <main+32>:   beq-    0x100016a0 <main+80>
0x10001674 <main+36>:   li      r0,10
0x10001678 <main+40>:   li      r9,20
0x1000167c <main+44>:   stw     r0,0(r3)
0x10001680 <main+48>:   stw     r9,4(r3)
0x10001684 <main+52>:   bl      0x10011be0 <shared_print>
0x10001688 <main+56>:   mr      r3,r31
0x1000168c <main+60>:   bl      0x10011c08 <shared_free>
0x10001690 <main+64>:   cmpwi   r3,0
```

[1] As of linux 2.6.9, the relevant code is found in fs/binfmt_elf.c

[2] The command to check is: objdump -s -j.interp testprint

[3] The source-code to ld.so lies in glibc source tree, elf/rtld.c

[4] The introduction of prelink has changed this since prelink will scan the system and calculate the relocation in a batch process rather than at load time.

[5] To see what ld.so thinks it is doing, running an application with environmental variable LD_DEBUG=all set.

```
0x10001694 <main+68>:   beq-    0x1000169c <main+76>
0x10001698 <main+72>:   li      r3,1
0x1000169c <main+76>:   bl      0x10011bf8 <exit>
0x100016a0 <main+80>:   lis     r3,4096
0x100016a4 <main+84>:   lis     r4,4096
```

The jump target is also a list of jump targets, which is called the GOT. Exact details are different, but on powerpc, they are initially entries to set an identifier for the function entry on r11 register, and jump to the dynamic linker. The dynamic linker will resolve the symbol and write a branch instruction to that address of GOT. The next time the function call happens, the new branch command is used. The initial call costs at least three branches to call the function, but the next call will cost two jumps.

```
(gdb) break 13
Breakpoint 1 at 0x10001664: file testprint.c, line 13.
(gdb) run
Starting program: /home/dancer/cvscheckout/whole/shlib-demo/libshared/.libs/testprint
libshared 0.1

Breakpoint 1, main (argc=14, argv=0x30080674) at testprint.c:13
13   dat = shared_new();
(gdb) list
8  int main(int argc, char **argv)
9  {
10   shareddata* dat;
11
12   shared_version();
13   dat = shared_new();
14   assert (dat);
15   dat->flag1=10;
16   dat->flag2=20;
17   shared_print(dat);
```

```
0x10011bd8 <__assert_fail+0>:   li      r11,0
0x10011bdc <__assert_fail+4>:   b         0x10011bb0 <__JCR_LIST__+56>
0x10011be0 <shared_print+0>:    li      r11,4
0x10011be4 <shared_print+4>:    b         0x10011bb0 <__JCR_LIST__+56>
0x10011be8 <shared_new+0>:      li      r11,8
0x10011bec <shared_new+4>:      b         0x10011bb0 <__JCR_LIST__+56>
0x10011bf0 <__libc_start_main+0>:         li      r11,12
0x10011bf4 <__libc_start_main+4>:         b         0x10011b90 <__JCR_LIST__+24>
Dump of assembler code from 0x10011c00 to 0x10011cff:
0x10011c00 <shared_version+0>:  b         0xffaea60 <shared_version>
0x10011c04 <shared_version+4>:  b         0x10011bb0 <__JCR_LIST__+56>
0x10011c08 <shared_free+0>:     li      r11,24
0x10011c0c <shared_free+4>:     b         0x10011bb0 <__JCR_LIST__+56>
```

After executing the function, the slot is replaced with a direct branch to the target function.

```
(gdb) cont
Continuing.

Breakpoint 2, main (argc=268509264, argv=0x10012058) at testprint.c:14
14   assert (dat);
(gdb) disassemble 0x10011be0 0x10011bff
Dump of assembler code from 0x10011be0 to 0x10011bff:
```

```
0x10011be0 <shared_print+0>:    li      r11,4
0x10011be4 <shared_print+4>:    b         0x10011bb0 <__JCR_LIST__+56>
0x10011be8 <shared_new+0>:      b         0xffaeb14 <shared_new>
0x10011bec <shared_new+4>:      b         0x10011bb0 <__JCR_LIST__+56>
0x10011bf0 <__libc_start_main+0>:         li      r11,12
0x10011bf4 <__libc_start_main+4>:         b        0x10011b90 <__JCR_LIST__+24>
0x10011bf8 <exit+0>:    li      r11,16
0x10011bfc <exit+4>:    b        0x10011bb0 <__JCR_LIST__+56>
End of assembler dump.
```

The relevant part that is jumped to before initialization is here:

```
0x10011bb0 <__JCR_LIST__+56>:   rlwinm  r12,r11,1,0,30
0x10011bb4 <__JCR_LIST__+60>:   add     r11,r12,r11
0x10011bb8 <__JCR_LIST__+64>:   li      r12,-19048
0x10011bbc <__JCR_LIST__+68>:   addis   r12,r12,4094
0x10011bc0 <__JCR_LIST__+72>:   mtctr   r12
0x10011bc4 <__JCR_LIST__+76>:   li      r12,0
0x10011bc8 <__JCR_LIST__+80>:   addis   r12,r12,12288
0x10011bcc <__JCR_LIST__+84>:   bctr
0x10011bd0 <__JCR_LIST__+88>:   .long 0x0
0x10011bd4 <__JCR_LIST__+92>:   .long 0x0
```

The binary that is linked to the library has a list of relocations, that will need to be filled up when the binary is loaded.

```
$ objdump -R testprint

.libs/testprint:     elf32-powerpc

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
10011b8c R_PPC_GLOB_DAT    __gmon_start__
10011bd8 R_PPC_JMP_SLOT    __assert_fail
10011be0 R_PPC_JMP_SLOT    shared_print
10011be8 R_PPC_JMP_SLOT    shared_new
10011bf0 R_PPC_JMP_SLOT    __libc_start_main
10011bf8 R_PPC_JMP_SLOT    exit
10011c00 R_PPC_JMP_SLOT    shared_version
10011c08 R_PPC_JMP_SLOT    shared_free
```

The shared library only holds a relative location; and their location needs to be calculated after loading.

```
$ objdump -t libshared.so| grep shared
libshared.so:  file format elf32-powerpc
00000000 l    df *ABS*  00000000       shared.c
00001aac g     F .text  00000068       shared_print
00001b14 g     F .text  00000040       shared_new
00001a60 g     F .text  00000048       shared_version
00001b54 g     F .text  00000038       shared_free
```

The memory map of loaded binary can be seen through `/proc/XXX/maps` file.

```
$ cat /proc/32396/maps
0ffad000-0ffaf000 r-xp 00000000 03:04 1160506    /work/libshared.so.0.0.0
```

```
0ffaf000-0ffbd000 ---p 00002000 03:04 1160506    /work/libshared.so.0.0.0
0ffbd000-0ffc0000 rwxp 00000000 03:04 1160506    /work/libshared.so.0.0.0
0ffd0000-0ffe6000 r-xp 00000000 03:04 162950  /lib/ld-2.3.2.so
0fff6000-0fff7000 rwxp 00016000 03:04 162950  /lib/ld-2.3.2.so
10000000-10002000 r-xp 00000000 03:04 1160514    /work/testprint
10011000-10012000 rwxp 00001000 03:04 1160514    /work/testprint
30000000-30002000 rw-p 30000000 00:00 0
3000d000-3013f000 r-xp 00000000 03:04 162953  /lib/libc-2.3.2.so
3013f000-3014d000 ---p 00132000 03:04 162953  /lib/libc-2.3.2.so
3014d000-3015a000 rwxp 00130000 03:04 162953  /lib/libc-2.3.2.so
3015a000-3015c000 rwxp 3015a000 00:00 0
7fffe000-80000000 rwxp 7fffe000 00:00 0
```

Shared library symbols originally look like:

```
(gdb) disassemble shared_free
Dump of assembler code for function shared_free:
0x0ffaeb54 <shared_free+0>:     cmpwi   r3,0
0x0ffaeb58 <shared_free+4>:     stwu    r1,-32(r1)
0x0ffaeb5c <shared_free+8>:     mflr    r0
0x0ffaeb60 <shared_free+12>:    li      r9,1
0x0ffaeb64 <shared_free+16>:    stw     r0,36(r1)
0x0ffaeb68 <shared_free+20>:    beq-    0xffaeb78 <shared_free+36>
0x0ffaeb6c <shared_free+24>:    crclr   4*cr1+eq
0x0ffaeb70 <shared_free+28>:    bl      0xffbf19c <__JCR_LIST__+128>
0x0ffaeb74 <shared_free+32>:    li      r9,0
0x0ffaeb78 <shared_free+36>:    lwz     r0,36(r1)
0x0ffaeb7c <shared_free+40>:    mr      r3,r9
0x0ffaeb80 <shared_free+44>:    addi    r1,r1,32
0x0ffaeb84 <shared_free+48>:    mtlr    r0
0x0ffaeb88 <shared_free+52>:    blr
```

Chapter 12

Credits and notes on this document

This document created: 10 Jan 2002, 13 Jan 2002. Junichi Uekawa (dancer@debian.org <mailto:dancer@debian.org>). Revised to be up-to-date on 17 Mar 2002. 31 Mar 2002, merged suggestions from David Schmitt. Translation to DocBook SGML done on 8 April 2002. Major spell-checking and review was done on 11 April 2002. Reorganisation of some text was done on 14 April 2002. Added more real-ish examples on what files go where, on 9 May 2002. Overview of usage of term SONAME was done on 12 May 2002, and they were corrected. Some more examples were added on 20 May 2002, and some language fixing was done on 25 May 2002. Added some notes on shlibs.local file at the end on 23 June 2002, because this document was missing that large portion. Added notes on libvorbis breakage on 8 Aug 2002, and start adding symbol versioning information (not really complete.) Notes on binary compatibility and shared library versioning are updated. Rewording on confusing binarycompat section cleared up slightly, 9 Aug 2002. Updates on versioned symbols, 13 Feb 2003.

More documentation on versioned symbols and dlopening problems, summarising what was discussed over libssl0.9.6/0.9.7 discussion, on 17 Mar 2003. Also, on that occasion, a read-through and general improvements was done over the text.

25 Jun 2003; Received fixes on some text from John Belmonte.

28 Sep 2003; added URL from what was said on IRC with regards to symbol versioning.

18 Apr 2004; updates on testing/unstable migration problem.

1 June 2004: update after vorlon's talk on dependency hells, a howto on making a shared library with versioned symbol.

14 Jul 2004: Quickly added a -rpath chapter, since nothing was said in libpkg-guide, and DWN had something to say.

5 Feb 2005: translated to docbook XML format

28 May 2005: added sed snippet to obtain shared library package name.

16 June 2005: Moved around text for how shared library should be named, from what files are installed. It was confusing.

17 June 2005: spelling mistake fix reported by Lior Kaplan.

25 June 2005: added reference to example of when SONAME doesn't change, and a >= relationship is required.

8 Jul 2005: addition of reference to how shared libraries get loaded, and other details.

15 Jul 2005: Remove the part referencing about dlopen module and -DEV dependency. -DEV dependency does not really help with dlopen modules with mixed symbols. The problems with dlopening is discussed in its own chapter.

12 Nov 2005: fix missing space, thanks to Nico Golde.

27 Nov 2005: typo fixes and grammar corrections, 'there is no dh_shlibs command, but dh_makeshlibs', thanks to Florian Ernst. Add reference to writing DSO's document from Ulrich Drepper.

3 Dec 2005: ”Table 8.1, slang-utf8 reference URL was not clickable”, ”Table 9.1 Changes and effects has duplicate row about Adding a function” reported by ’Davide’

5 Mar 2006: reformatting to make a PDF version readable.

20 May 2006: reflected result of discussing the document over with Josselin Mouette, over his presentation on shared libraries at Debconf in Mexico. Majorly reorganised text around, so that the chapters are in a more meaningful order. Unify example shared library name to ”libfoo” and ”libbar”.

31 Jul 2006: fix typo.