

# Packaging shared libraries

Josselin MOUETTE — CS SYSTÈMES D'INFORMATION

6th April 2006

## 1 Introducing shared libraries

### 1.1 Basic concepts

A library is a piece of code that can be used in several binaries, split out for factorization reasons. In the old days, libraries were all *statically linked*; that is, they were included directly in the resulting binary. Modern operating systems use shared libraries, which are compiled in separate files and loaded together with the binary at startup time. Shared libraries are the most widespread use of *shared objects*, files containing code that can be loaded at runtime, generally with the `.so` extension.

### 1.2 A bit of terminology

**API** The *Application Programming Interface* of a library describes how it can be used by the programmer. It generally consists in a list of structures and functions and their associated behavior. Changing the behavior of a function or the type of arguments it requires *breaks* the API: programs that used to compile with an older version of the library will stop building.

**ABI** The *Application Binary Interface* defines the low-level interface between a shared library and the binary using it. It is specific to the architecture and the operating system, and consists in a lists of *symbols* and their associated type and behavior. A binary linked to a shared library will be able to run with another library, or another version of that library, provided that it implements the same ABI. Adding elements to a structure or turning a function into a macro *breaks* the ABI: binaries that used to run with an older version of the library will stop loading. Most of the time, breaking the API also breaks the ABI.

**SONAME** The "SONAME" is the canonical name of a shared library, defining an ABI for a given operating system and architecture. It is defined when building the library. The convention for SONAMEs is to use `libfoo.so.N` and to increment N whenever the ABI is changed. This way, ABI-incompatible versions of the library and binaries using them can coexist on the same system.

### 1.3 Linking and using libraries

A simple example of building a library using `gcc` :

```
gcc -fPIC -c -o foo-init.o foo-init.c
[ ... ]
gcc -shared -Wl,-soname,libfoo.so.3 -o libfoo.so.3 foo-init.o foo-client.o [...]
ln -s libfoo.so.3 libfoo.so
```

As the command line shows, the SONAME is defined at that time. The symbolic link is needed for compilation of programs using the library. Supposing it has been installed in a standard location, you can link a binary — which can be another shared library — using it with `-lfoo`. The linker looks for `libfoo.so`, and stores the SONAME found (`libfoo.so.3`) in the binary's ELF<sup>1</sup> header.

The output of the `objdump -p` command shows the headers of an ELF object. For the library, the output contains:

---

<sup>1</sup>*Executable and Linking Format*: the binary format for binaries and shared objects on most UNIX systems.

```
SONAME      libfoo.so.3
```

For the binary, it contains:

```
NEEDED      libfoo.so.3
```

The symbols provided by the library remain undefined in the binary at that time. In the dynamic symbol table showed by `objdump -T`, the library contains the symbol:

```
0807c8e0 g DF .text 0000007d Base      foo_init
```

while in the binary it remains undefined:

```
00000000 DF *UND* 0000001c          foo_init
```

When the binary is started, the GNU *dynamic linker*<sup>2</sup> looks for the NEEDED sections and loads the libraries listed there, using the SONAME as a file name. It then maps the undefined symbols to the ones found in the libraries.

## 1.4 Libtool

Libtool is a tool designed to simplify the build process of libraries. It is full of features that make the developers' life easier, and full of bugs that bring added complexity for system administrators and especially distribution maintainers. Its paradigm is to build an extra file, named `libfoo.la`, which contains some metadata about the library; most importantly, the list of library dependencies for the library itself. Together with this file, it can build the shared version `libfoo.so` and the static version `libfoo.a` of the library.

It integrates easily with `autoconf` and `automake`. You can put in the `configure.ac`<sup>3</sup>:

```
AM_PROG_LIBTOOL
VERSION_INFO=3:1:0
AC_SUBST(VERSION_INFO)
```

and in the `Makefile.am`:

```
libfoo_la_SOURCES = foo-init.c foo-client.c foo.h [...]
libfoo_la_LDFLAGS = -version-info @VERSION_INFO@
libfoo_HEADERS = foo.h
```

## 1.5 Pkgconfig

Pkgconfig is a tool to replace the variety of `libfoo-config` scripts in a standard way that integrates with `autoconf`. Here is a sample file, `libnautilus-burn.pc`:

```
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include/libnautilus-burn
```

```
Name: libnautilus-burn
Description: Nautilus Burn Library
Version: 2.12.3
Requires: glib-2.0 gtk+-2.0
Libs: -L${libdir} -lnautilus-burn
Cflags: -I${includedir}
```

The `Cflags:` and `Libs:` fields provide the list of `CFLAGS` and `LDFLAGS` to use for linking with that library. The `Requires:` field provides some dependencies that a binary using that library should also link with. In this case, `pkgconfig` will also look for `glib-2.0.pc` and `gtk+-2.0.pc`.

Integration with `autoconf` is provided. Here is an example `configure.ac` test for a package requiring the `GTK+` library:

---

<sup>2</sup>Other linkers can use a different scheme, especially when it comes to filename lookup.

<sup>3</sup>The version information is given for libtool's versioning scheme. You can read more about it in the libtool manual.

```
PKG_CHECK_MODULES(GTK, gtk+-2.0 >= 2.6.0,,
                  AC_MSG_ERROR([GTK+-2.0 is required]))
```

## 2 Debian packaging of a shared library

### 2.1 Simple case – what the policy mandates

Packaging a simple library for Debian is not much different from another piece of software. In all cases there should at least be two packages:

- `libfoo3`, containing the `/usr/lib/*.so.*` files, so that you get `libfoo.so.3`. The `postinst` script of this package should contain a call to the `ldconfig` command, and it has to be registered in `dpkg`'s *shlibs* database. This can be achieved by a call to `dh_makeshlibs`.
- `libfoo-dev` or `libfoo3-dev`, containing the headers in `/usr/include`, and other files in `/usr/lib`: the `libfoo.so` symbolic link, the `libfoo.a` static library, and if relevant `libfoo.la` (in `/usr/lib`) and `libfoo.pc` (in `/usr/share/pkgconfig`<sup>4</sup>). It should depend on `libfoo3` (= `$Source-Version`).

The *shlibs* system provides a mapping of library SONAMES to package names and minimal versions for the ABIs a of libraries a package is built against.

### 2.2 Updating the package

As for anything providing an interface, shared libraries have to be treated carefully when it comes to updating the package.

- If the ABI has not changed at all, no changes are required to the package.
- The most common case is the ABI being changed in a backwards-compatible way, by adding symbols. In this case, the *shlibs* system should be informed of the minimum version required. This is achieved by changing the `rules` file to call:

```
dh_makeshlibs -V'libfoo3 (>= 3.1.0)'
```

The referenced version is the one of the latest version where the ABI was changed.

- When some symbols are removed or their meaning is changed, the ABI is broken and the SONAME should have changed. The shared library package name has to be changed to reflect this new SONAME: `libfoo3` becomes `libfoo4`.
- If the API changes, some packages using the library may stop building. If the change is small, it may only require fixing of a handful of packages. If it's a broad change, the simplest course of action is to change the development package name: `libfoo3-dev` becomes `libfoo4-dev`.

### 2.3 Library transitions

Whenever the ABI is broken, a library transition starts. Before anything like this happens, the release team should be asked for approval, so that they know the transition will happen. If possible, two transition implicating the same packages should be avoided, as they would have to complete together.

All packages using the library have to be rebuilt in the *unstable* distribution so that they can go to *testing* together. Depending on the library, the optimal course of action may vary.

- If there is a small enough number of reverse dependencies, things can go fast: an upload right to *unstable*, asking the release team to trigger a set of binary NMUs for all depending packages.
- More complex cases, especially if some reverse dependencies can fail to build, should be started in *experimental*.
- For some nightmare libraries, several source versions are present at once, even in stable releases. The examples of `gnutls` and `libpng` come to mind.

---

<sup>4</sup>Pkgconfig has started moving its `.pc` files from `/usr/lib/pkgconfig` and this should be encouraged.

## 2.4 Providing a debugging version

If the library is known to cause crashes or is under development, the availability of debugging symbols is quite helpful. Fortunately, `debhelper` can do all of this automatically. After defining an empty `libfoo3-dbg` package, the magic command is:

```
dh_strip --dbg-package=libfoo3-dbg
```

This will move debugging symbols in `/usr/lib/debug` in this package; debuggers like `gdb` can use them automatically.

## 2.5 More complex cases – how to avoid circular dependencies

With large and complex libraries, other kinds of issues appear. Considering the example of `gconf2`, the upstream distribution contains:

- a library used by applications,
- a per-user daemon,
- chunks of data, mostly localization files,
- configuration files,
- documentation,
- support binaries using the library.

To avoid having in `libgconf2-4` any files outside versioned directories, the configuration and data were moved to a `gconf2-common` package. Documentation was put in `libgconf2-dev`, where it is useful, and as mandated by policy, support binaries were put in a separate package, named `gconf2`.

The tricky part is the daemon. When it is not running for the user, it is started by the application using the GConf library, which means the library should depend on the daemon. Still, the daemon is linked with the library. Until 2005, the daemon was in the `gconf2` package, meaning a *circular dependency* between `gconf2` and `libgconf2-4`.

Circular dependencies lead to various issues:

- APT randomly fails to upgrade such packages in large-scale upgrades;
- the `postinst` scripts are executed in a random order;
- worst of all, the `preRM` scripts of depending packages can be executed while dependent packages have been removed. This issue turned out to be a release-critical bug for `gconf2`, seriously breaking the build daemons' environment.

The solution to circular dependencies is to put files depending on each other in a single package: if they cannot live without each other, there is no reason to put them in separate packages. Thus, the daemon was put in the `libgconf2-4` package. To avoid including non-versioned files in the library package, which can be an issue in case of a SONAME change and which will become an issue for the multiarch project, the packaging was modified to use `/usr/lib/libgconf2-4` as its *libexecdir*, putting the daemon in this directory.

Despite having been tested in *experimental*, no less than 6 new RC bugs were reported against the new package. If anything, it means such changes have to be done with extreme care, thinking of all upgrade scenarios; *unstable* users can imagine unsought ways to torture APT and will install any package combination that is allowed.

## 3 Common developer mistakes

A commonly spread game among upstream library developers is to keep Debian developers busy. Here are some common ways for them to achieve this goal.

### 3.1 Non-PIC code

As a shared library can be loaded at any position in the address space, its compiled code cannot contain anything that depends on that absolute position. The compiler has to be instructed to build *Position Independent Code* with the `-fPIC` option. Usually, this means building two versions of each code object, one with `-fPIC` and one without. Libtool will do this automatically.

However, some developers using their own build system will forget this flag. Most of the time, they only work with the `i386`, on which non-PIC shared libraries still work. Furthermore, PIC code is slower on this architecture, as it is missing a relative jump instruction, getting some performance fanatics to knowingly remove it.

Non-PIC code can also arise from inline assembly code, if it was not written with position independence in mind. In all cases, lintian will emit an error when finding non-PIC code, which shows up as a `TEXTREL` section in the output of `objdump -p`.

### 3.2 Unstable ABI without SONAME changes

Sometimes, an ABI change is noticed in a released library without a SONAME change. Removal or change of generally unused symbols is the most common case. In such cases, upstream developers will generally not change the SONAME of the library and distributors have to deal with it. The solution is to change the package name, `libfoo3` becoming `libfoo3a`. The new package has to conflict with the old one and all depending packages have to be rebuilt.

Some upstream library developers go even further, not having a clue about what is an ABI. They consider the shared library just like the static version and the ABI can change at each release. Examples include `hdf5` or the Mozilla suite. In case of such an unstable ABI, a simple course of action is to ship only a static version of the library. However, it makes the security team's work a nightmare, as every package using the library has to be rebuilt after a security update.

A more clever solution to such breakage is to give a Debian-specific SONAME to the library and to change it whenever needed. This work has been done for the Mozilla suite in the `xulrunner` package. When the breakage is systematic as in `hdf5`, the change can be automated with libtool, as shows this sample from the diff file:

```
-LT_LINK_LIB=$(LT) --mode=link $(CC) -rpath $(libdir) $(DYNAMIC_DIRS)
+LT_LINK_LIB=$(LT) --mode=link $(CC) -rpath $(libdir) -release $(H5_VERSION) -version-info 0
```

The `-release` flag for libtool gives a string to add to the library name. Thus, the `libhdf5.so.0` library becomes `libhdf5-1.6.5.so.0`.

As for the build process, the library package name has to be changed for each new upstream version: here it becomes `libhdf5-1.6.5-0`. Automated `debian/control` generation helps making updates as easy as with other packages — apart from the fact they have to go through the *NEW* queue at every upstream release.

It should be noted that a clever library design can eliminate most causes for an ABI breakage. An example of such a design can be found in GNOME libraries: all data structures are hidden in private structures that cannot be found in public headers, and they are only accessible through helper functions that always answer to a functional need. Most GNOME libraries haven't changed their SONAMES for several years despite major architectural changes.

### 3.3 Exporting private symbols

At link time, all functions and global variables that were not declared as `static` in the source code become exported symbols in the generated library. That includes functions that do not appear in public header files, and which as such should not be used as part of the API.

Some application developers make use of this small hole. They define the prototype of these private functions in their own headers and make use of them at link time. Such an application is heavily buggy, as it will break when the library developers decide to change their private functions. To detect these applications reliably and to prevent them from running at all, the list of exported symbols should be restricted. It also helps avoiding symbol name conflicts between libraries.

It can be achieved using a simple version script (see p. 6). There is also a feature from libtool which allows to automate this process. Here is a sample taken from the `SDL_mixer` Makefile.am file:

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

This way, only symbols being part of the `SDL_mixer` namespace, those beginning with `Mix_`, are exported.

Namespace conflicts can also occur between symbols from the library itself and functions belonging to a program linking to it. The ELF architecture allows a program to override function definitions from a shared library. The symbols can be protected against this kind of override by using the `-Wl,-Bsymbolic` argument at link time. It should be used for libraries exporting too generic functions, and it should be systematically applied to library plugins, *e.g.* GTK+ input methods or theme engines. Such plugins can have their code intermixed with any kind of application that has not been tested with them, and namespace conflicts should be avoided in this case.

## 4 Going further – reducing the release team’s hair loss

### 4.1 Versioning the symbols

#### 4.1.1 The problem

Let’s consider the following simple scenario: a picture viewer written using GTK+. The software makes use of `libgtk` for its graphical interface, and of `libpng` to load PNG images. However, `libgtk` by itself already depends on `libpng`. When the ABI of `libpng` changed, and `libpng.so.2` became `libpng.so.3`, both GTK+ and the application had to be rebuilt. In this kind of case, if only the picture viewer is rebuilt, it will end up depending indirectly on both `libpng.so.2` and `libpng.so.3`.

Here, the software is faced with a design flaw in the dynamic linker: when resolving library dependencies, all symbols found in all dependencies, direct or indirect, are loaded in a global symbol table. Once this is done, there is no way to tell between a symbol that comes from `libpng.so.2` and one with the same name coming from `libpng.so.3`. This way, GTK+ can call some functions that belong to `libpng.so.3` while using the ABI from `libpng.so.2`, causing crashes.

#### 4.1.2 The solution

Such issues can be solved by introducing *versioned symbols* in the libraries. Another option has to be passed at link time:

```
libpng12_la_LDFLAGS += -Wl,--version-script=libpng.vers
```

The *version script* referenced here can be a simple script to give the same version to all symbols:

```
PNG12_0 {
*; };
```

The 1.2.x version (`libpng.so.3`) is given the `PNG12_0` version, while the 1.0.x version is given `PNG10_0`. Let’s have a look at the symbols in the libraries using the `objdump -T` command. For the 1.0.x version we have:

```
00006260 g DF .text 00000011 PNG10_0 png_init_io
```

and for the 1.2.x version:

```
000067a0 g DF .text 00000011 PNG12_0 png_init_io
```

Now, when a binary is linked against this new version, it still marks the symbols from `libpng` as undefined, but with a symbol version:

```
00000000 DF *UND* 00000011 PNG12_0 png_init_io
```

When two symbols with the same name are available in the global symbol time, the dynamic linker will know which one to use.

### 4.1.3 Caveats

To benefit from versioned symbols, all packages using the library have to be rebuilt. Once this is done, it is possible to migrate from a library version to another providing the same symbols, transparently. For a library as widely used as libpng, this was a very slow transition mechanism. Before the *sarge* release, all packages using libpng have been rebuilt using these versioned symbols, whether using version 1.0.x or 1.2.x. After the release, the 1.0.x version has been entirely removed, and packages using 1.0.x have migrated to 1.2.x without major issues. Having waited for a stable release allows to be sure upgrades across stable releases go smoothly.

It is of critical importance to forward such changes to upstream developers and to make sure they are adopted widely. Otherwise, if upstream developers or another distributor chooses to introduce a *different* version for these symbols, the two versions of the library become incompatible. A recent example is found with `libmysqlclient`: the patch was accepted by upstream developers, but they choose to change the symbols version, without knowing it would render the binary library incompatible with the one Debian had been shipping.

### 4.1.4 Improving the version script

In the case of libpng, it is also beneficial to restrict the list of exported symbols. All of this can be done in a single version script which is automatically generated from the headers:

```
PNG12_0 { global:
png_init_io;
png_read_image;
[...]
local: *; };
```

## 4.2 Restricting the list of dependencies

### 4.2.1 Relibtoolizing packages

As explained p. 2, libtool stores the list of dependencies of a library in the `libfoo.la` file. While they are only useful for static linking (as the `libfoo.a` file does not store its dependencies), it also uses them for dynamic linking. When the dependencies are also using libtool, it will recurse through `.la` files looking for all dependencies.

As a result, binaries end up being direct linked with many libraries they do not actually require. While this is harmless on a stable platform, it can cause major issues with a system continuously under development like Debian, as dependencies are continuously evolving, being added, removed or migrated to new versions. These unneeded dependencies result in unneeded rebuilds during library transitions and added complexity for migration to the *testing* distribution.

The Debian `libtool` package contains a patch that corrects this behavior. However, as libtool only produces scripts that get included with the upstream package, the package acted upon has to include as a patch the result of a *relibtoolization* using the Debian version of libtool:

```
libtoolize --force --copy ; aclocal ; automake --force-missing --add-missing --foreign --copy ;
autoconf ; rm -rf autom4te.cache
```

It has the drawback to add some continuous burden on the Debian maintainer, as it needs to be done for each new upstream release. Furthermore, it is generally not enough, as indirect dependencies can be added by other sources in a complex build process.

When recursing through dependencies, libtool also adds them to the list of dependencies of the library it's building. For example, when building `libfoo` which requires `libbar` which it turn depends on `libbaz`, it will add a reference to `libbaz` in `libfoo.la`. If the dependency on `libbaz` is removed, packages depending on `libfoo` will fail to build, as they will look for a library that does not exist anymore.

### 4.2.2 Pkgconfig

Another widespread source of indirect dependencies is `pkgconfig`. As it also handles dependencies through `Requires:` fields, it will link the binary with several indirect dependencies. Furthermore, developers often add some indirect dependencies in `Libs:` fields.

Recent changes in pkgconfig allow the use of `Requires.private:` and `Libs.private` fields. These libraries and dependencies will be linked in only when using static linking. Here is an example in `cairo.pc`:

```
Requires.private: freetype2 >= 8.0.2 fontconfig xrender libpng12
```

Unlike the relibtoolization, these changes have to be made in the packages that are depended upon, not in the package that hits the problem. Furthermore, it has been argued that libraries that have their headers automatically included (like glib when using GTK+) should be linked in by default nevertheless.

### 4.2.3 GNU linker magic

The GNU linker has an option that can make all indirect dependencies go away: `--as-needed`. For example, it can be passed to the configure script:

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

When passed this option, the dynamic linker does not necessarily make the binary it is linking depend on the shared libraries passed with `-lfoo` arguments. First, it checks that the binary is actually using some symbols in the library, skipping the library if not needed. This mechanism dramatically reduces the list of unneeded dependencies, including the ones upstream developers could have explicitly added.

This option should not be used blindly. In some specific cases, the library should be linked in even when none of its symbols are used. Support for it is still young, and it should not be considered 100 % reliable. Furthermore, it does not solve the issue of libtool recursing in `.la` files and searching for removed libraries.

To make things worse, a recent change in libtool introduced argument reordering at link time, which turns the `--as-needed` option into a dummy one. This only happens when building libraries, not applications. A workaround was developed, as a patch for `ltmain.sh`, for the `libgnome` package where it is of large importance. It is currently waiting for a cleanup before being submitted as another Debian-specific libtool change<sup>5</sup>.

## Conclusion

Apart from treating each update with care, there is no general rule for packaging shared libraries. There are many solutions and workarounds for known problems, but each of them adds complexity to the packaging and should be considered on a case-by-case basis. As the long list of problems shows, being release manager is not an easy task, and library package maintainers should do their best to keep the release team's task feasible.

There is a huge number of software libraries distributed in the wild, and almost two thousand of them are shipped in the Debian distribution. Among all developers of these libraries, many of them are not aware of shared libraries specific issues. The Debian maintainer's job is more than working around these issues: it is to help upstream understand them and fix their packages. As such, forwarding and explaining patches is a crucial task.

---

<sup>5</sup>The upstream libtool developers have stated it may be fixed in the future, but not even in libtool 2.0.