7 Library interface versions

The most difficult issue introduced by shared libraries is that of creating and resolving runtime dependencies. Dependencies on programs and libraries are often described in terms of a single name, such as `sed`. So, one may say "libtool depends on sed," and that is good enough for most purposes.

However, when an interface changes regularly, we need to be more specific: "Gnus 5.1 requires Emacs 19.28 or above." Here, the description of an interface consists of a name, and a "version number."

Even that sort of description is not accurate enough for some purposes. What if Emacs 20 changes enough to break Gnus 5.1?

The same problem exists in shared libraries: we require a formal version system to describe the sorts of dependencies that programs have on shared libraries, so that the dynamic linker can guarantee that programs are linked only against libraries that provide the interface they require.

7.1 What are library interfaces?

Interfaces for libraries may be any of the following (and more):

- global variables: both names and types
- global functions: argument types and number, return types, and function names
- standard input, standard output, standard error, and file formats
- sockets, pipes, and other inter-process communication protocol formats

Note that static functions do not count as interfaces, because they are not directly available to the user of the library.

7.2 Libtool's versioning system

Libtool has its own formal versioning system. It is not as flexible as some, but it is definitely the simplest of the more powerful versioning systems.

Think of a library as exporting several sets of interfaces, arbitrarily represented by integers. When a program is linked against a library, it may use any subset of those interfaces.

Libtool's description of the interfaces that a program uses is simple: it encodes the least and the greatest interface numbers in the resulting binary (*first-interface*, *last-interface*).

The dynamic linker is guaranteed that if a library supports *every* interface number between *first-interface* and *last-interface*, then the program can be relinked against that library.

Note that this can cause problems because libtool's compatibility requirements are actually stricter than is necessary.

Say '`libhello`' supports interfaces 5, 16, 17, 18, and 19, and that libtool is used to link '`test`' against '`libhello`'.

Libtool encodes the numbers 5 and 19 in '`test`', and the dynamic linker will only link '`test`' against libraries that support *every* interface between 5 and 19. So, the dynamic linker refuses to link '`test`' against '`libhello`'!

In order to eliminate this problem, libtool only allows libraries to declare consecutive interface numbers. So, '`libhello`' can declare at most that it supports interfaces 16 through 19. Then, the dynamic linker will link '`test`' against '`libhello`'.

So, libtool library versions are described by three integers:

current	The most recent interface number that this library implements.
revision	The implementation number of the *current* interface.
age	The difference between the newest and oldest interfaces that this library implements. In other words, the library implements all the interface numbers in the range from number `current - age` to `current`.

If two libraries have identical *current* and *age* numbers, then the dynamic linker chooses the library with the greater *revision* number.

7.3 Updating library version information

If you want to use libtool's versioning system, then you must specify the version information to libtool using the '`-version-info`' flag during link mode (see Section 4.2 [Link mode], page 16).

This flag accepts an argument of the form '`current[:revision[:age]]`'. So, passing '`-version-info 3:12:1`' sets *current* to 3, *revision* to 12, and *age* to 1.

If either *revision* or *age* are omitted, they default to 0. Also note that *age* must be less than or equal to the *current* interface number.

Here are a set of rules to help you update your library version information:

1. Start with version information of '`0:0:0`' for each libtool library.
2. Update the version information only immediately before a public release of your software. More frequent updates are unnecessary, and only guarantee that the current interface number gets larger faster.
3. If the library source code has changed at all since the last update, then increment *revision* ('`c:r:a`' becomes '`c:`$r+1$`:a`').
4. If any interfaces have been added, removed, or changed since the last update, increment *current*, and set *revision* to 0.
5. If any interfaces have been added since the last public release, then increment *age*.
6. If any interfaces have been removed since the last public release, then set *age* to 0.

Never try to set the interface numbers so that they correspond to the release number of your package. This is an abuse that only fosters misunderstanding of the purpose of library versions. Instead, use the '`-release`' flag (see Section 7.4 [Release numbers], page 38), but be warned that every release of your package will not be binary compatible with any other release.

7.4 Managing release information

Often, people want to encode the name of the package release into the shared library so that it is obvious to the user which package their programs are linked against. This convention is used especially on GNU/Linux:

```
trick$ ls /usr/lib/libbfd*
/usr/lib/libbfd.a           /usr/lib/libbfd.so.2.7.0.2
/usr/lib/libbfd.so
trick$
```

On '`trick`', '`/usr/lib/libbfd.so`' is a symbolic link to '`libbfd.so.2.7.0.2`', which was distributed as a part of '`binutils-2.7.0.2`'.

Unfortunately, this convention conflicts directly with libtool's idea of library interface versions, because the library interface rarely changes at the same time that the release number does, and the library suffix is never the same across all platforms.

So, in order to accommodate both views, you can use the '`-release`' flag in order to set release information for libraries for which you do not want to use '`-version-info`'. For the '`libbfd`' example, the next release that uses libtool should be built with '`-release 2.9.0`', which will produce the following files on GNU/Linux:

```
trick$ ls /usr/lib/libbfd*
/usr/lib/libbfd-2.9.0.so     /usr/lib/libbfd.a
/usr/lib/libbfd.so
trick$
```

In this case, '`/usr/lib/libbfd.so`' is a symbolic link to '`libbfd-2.9.0.so`'. This makes it obvious that the user is dealing with '`binutils-2.9.0`', without compromising libtool's idea of interface versions.

Note that this option causes a modification of the library name, so do not use it unless you want to break binary compatibility with any past library releases. In general, you should only use '`-release`' for package-internal libraries or for ones whose interfaces change very frequently.