GNU Libtool

For version 1.4.2, 10 September 2001

Gordon Matzigkeit
Alexandre Oliva
Thomas Tanner
Gary V. Vaughan

1 Introduction

In the past, if a source code package developer wanted to take advantage of the power of shared libraries, he needed to write custom support code for each platform on which his package ran. He also had to design a configuration interface so that the package installer could choose what sort of libraries were built.

GNU Libtool simplifies the developer's job by encapsulating both the platform-specific dependencies, and the user interface, in a single script. GNU Libtool is designed so that the complete functionality of each host type is available via a generic interface, but nasty quirks are hidden from the programmer.

GNU Libtool's consistent interface is reassuring... users don't need to read obscure documentation in order to have their favorite source package build shared libraries. They just run your package `configure` script (or equivalent), and libtool does all the dirty work.

There are several examples throughout this document. All assume the same environment: we want to build a library, '`libhello`', in a generic way.

'`libhello`' could be a shared library, a static library, or both... whatever is available on the host system, as long as libtool has been ported to it.

This chapter explains the original design philosophy of libtool. Feel free to skip to the next chapter, unless you are interested in history, or want to write code to extend libtool in a consistent way.

1.1 Motivation for writing libtool

Since early 1995, several different GNU developers have recognized the importance of having shared library support for their packages. The primary motivation for such a change is to encourage modularity and reuse of code (both conceptually and physically) in GNU programs.

Such a demand means that the way libraries are built in GNU packages needs to be general, to allow for any library type the package installer might want. The problem is compounded by the absence of a standard procedure for creating shared libraries on different platforms.

The following sections outline the major issues facing shared library support in GNU, and how shared library support could be standardized with libtool.

The following specifications were used in developing and evaluating this system:

1. The system must be as elegant as possible.
2. The system must be fully integrated with the GNU Autoconf and Automake utilities, so that it will be easy for GNU maintainers to use. However, the system must not require these tools, so that it can be used by non-GNU packages.
3. Portability to other (non-GNU) architectures and tools is desirable.

1.2 Implementation issues

The following issues need to be addressed in any reusable shared library system, specifically libtool:

1. The package installer should be able to control what sort of libraries are built.
2. It can be tricky to run dynamically linked programs whose libraries have not yet been installed. `LD_LIBRARY_PATH` must be set properly (if it is supported), or programs fail to run.
3. The system must operate consistently even on hosts which don't support shared libraries.
4. The commands required to build shared libraries may differ wildly from host to host. These need to be determined at configure time in a consistent way.
5. It is not always obvious with which suffix a shared library should be installed. This makes it difficult for '`Makefile`' rules, since they generally assume that file names are the same from host to host.
6. The system needs a simple library version number abstraction, so that shared libraries can be upgraded in place. The programmer should be informed how to design the interfaces to the library to maximize binary compatibility.
7. The install '`Makefile`' target should warn the package installer to set the proper environment variables (`LD_LIBRARY_PATH` or equivalent), or run `ldconfig`.

1.3 Other implementations

Even before libtool was developed, many free software packages built and installed their own shared libraries. At first, these packages were examined to avoid reinventing existing features.

Now it is clear that none of these packages have documented the details of shared library systems that libtool requires. So, other packages have been more or less abandoned as influences.

1.4 A postmortem analysis of other implementations

In all fairness, each of the implementations that were examined do the job that they were intended to do, for a number of different host systems. However, none of these solutions seem to function well as a generalized, reusable component.

Most were too complex to use (much less modify) without understanding exactly what the implementation does, and they were generally not documented.

The main difficulty is that different vendors have different views of what libraries are, and none of the packages which were examined seemed to be confident enough to settle on a single paradigm that just *works*.

Ideally, libtool would be a standard that would be implemented as series of extensions and modifications to existing library systems to make them work consistently. However, it is not an easy task to convince operating system developers to mend their evil ways, and people want to build shared libraries right now, even on buggy, broken, confused operating systems.

For this reason, libtool was designed as an independent shell script. It isolates the problems and inconsistencies in library building that plague 'Makefile' writers by wrapping the compiler suite on different platforms with a consistent, powerful interface.

With luck, libtool will be useful to and used by the GNU community, and that the lessons that were learned in writing it will be taken up by designers of future library systems.

2 The libtool paradigm

At first, libtool was designed to support an arbitrary number of library object types. After libtool was ported to more platforms, a new paradigm gradually developed for describing the relationship between libraries and programs.

In summary, "libraries are programs with multiple entry points, and more formally defined interfaces."

Version 0.7 of libtool was a complete redesign and rewrite of libtool to reflect this new paradigm. So far, it has proved to be successful: libtool is simpler and more useful than before.

The best way to introduce the libtool paradigm is to contrast it with the paradigm of existing library systems, with examples from each. It is a new way of thinking, so it may take a little time to absorb, but when you understand it, the world becomes simpler.

3 Using libtool

It makes little sense to talk about using libtool in your own packages until you have seen how it makes your life simpler. The examples in this chapter introduce the main features of libtool by comparing the standard library building procedure to libtool's operation on two different platforms:

'a23' An Ultrix 4.2 platform with only static libraries.

'burger' A NetBSD/i386 1.2 platform with shared libraries.

You can follow these examples on your own platform, using the preconfigured libtool script that was installed with libtool (see Section 5.3 [Configuring], page 20).

Source files for the following examples are taken from the 'demo' subdirectory of the libtool distribution. Assume that we are building a library, 'libhello', out of the files 'foo.c' and 'hello.c'.

Note that the 'foo.c' source file uses the `cos` math library function, which is usually found in the standalone math library, and not the C library (see section "Trigonometric Functions" in *The GNU C Library Reference Manual*). So, we need to add `-lm` to the end of the link line whenever we link 'foo.o' or 'foo.lo' into an executable or a library (see Chapter 8 [Inter-library dependencies], page 33).

The same rule applies whenever you use functions that don't appear in the standard C library... you need to add the appropriate `-lname` flag to the end of the link line when you link against those objects.

After we have built that library, we want to create a program by linking 'main.o' against 'libhello'.

3.1 Creating object files

To create an object file from a source file, the compiler is invoked with the '-c' flag (and any other desired flags):

```
burger$ gcc -g -O -c main.c
burger$
```

The above compiler command produces an object file, 'main.o', from the source file 'main.c'.

For most library systems, creating object files that become part of a static library is as simple as creating object files that are linked to form an executable:

```
burger$ gcc -g -O -c foo.c
burger$ gcc -g -O -c hello.c
burger$
```

Shared libraries, however, may only be built from *position-independent code* (PIC). So, special flags must be passed to the compiler to tell it to generate PIC rather than the standard position-dependent code.

Since this is a library implementation detail, libtool hides the complexity of PIC compiler flags by using separate library object files (which end in '.lo' instead of '.o'). On systems without shared libraries (or without special PIC compiler flags), these library object files are identical to "standard" object files.

To create library object files for 'foo.c' and 'hello.c', simply invoke libtool with the standard compilation command as arguments (see Section 4.1 [Compile mode], page 15):

```
a23$ libtool gcc -g -O -c foo.c
gcc -g -O -c foo.c
echo timestamp > foo.lo
a23$ libtool gcc -g -O -c hello.c
gcc -g -O -c hello.c
echo timestamp > hello.lo
a23$
```

Note that libtool creates two files for each invocation. The '.lo' file is a library object, which may be built into a shared library, and the '.o' file is a standard object file. On 'a23', the library objects are just timestamps, because only static libraries are supported.

On shared library systems, libtool automatically inserts the PIC generation flags into the compilation command, so that the library object and the standard object differ:

```
burger$ libtool gcc -g -O -c foo.c
gcc -g -O -c -fPIC -DPIC foo.c
mv -f foo.o foo.lo
gcc -g -O -c foo.c >/dev/null 2>&1
burger$ libtool gcc -g -O -c hello.c
gcc -g -O -c -fPIC -DPIC hello.c
mv -f hello.o hello.lo
gcc -g -O -c hello.c >/dev/null 2>&1
burger$
```

Notice that the second run of GCC has its output discarded. This is done so that compiler warnings aren't annoyingly duplicated.

3.2 Linking libraries

Without libtool, the programmer would invoke the `ar` command to create a static library:

```
burger$ ar cru libhello.a hello.o foo.o
burger$
```

But of course, that would be too simple, so many systems require that you run the `ranlib` command on the resulting library (to give it better karma, or something):

```
burger$ ranlib libhello.a
burger$
```

It seems more natural to use the C compiler for this task, given libtool's "libraries are programs" approach. So, on platforms without shared libraries, libtool simply acts as a wrapper for the system `ar` (and possibly `ranlib`) commands.

Again, the libtool library name differs from the standard name (it has a '.la' suffix instead of a '.a' suffix). The arguments to libtool are the same ones you would use to produce an executable named 'libhello.la' with your compiler (see Section 4.2 [Link mode], page 15):

```
a23$ libtool gcc -g -O -o libhello.la foo.o hello.o
libtool: cannot build libtool library 'libhello.la' from non-libtool \
                objects
```

```
a23$
```

Aha! Libtool caught a common error... trying to build a library from standard objects instead of library objects. This doesn't matter for static libraries, but on shared library systems, it is of great importance.

So, let's try again, this time with the library object files. Remember also that we need to add *-lm* to the link command line because 'foo.c' uses the cos math library function (see Chapter 3 [Using libtool], page 5).

Another complication in building shared libraries is that we need to specify the path to the directory in which they (eventually) will be installed (in this case, '/usr/local/lib')[1]:

```
a23$ libtool gcc -g -O -o libhello.la foo.lo hello.lo \
                -rpath /usr/local/lib -lm
mkdir .libs
ar cru .libs/libhello.a foo.o hello.o
ranlib .libs/libhello.a
creating libhello.la
a23$
```

Now, let's try the same trick on the shared library platform:

```
burger$ libtool gcc -g -O -o libhello.la foo.lo hello.lo \
                -rpath /usr/local/lib -lm
mkdir .libs
ld -Bshareable -o .libs/libhello.so.0.0 foo.lo hello.lo -lm
ar cru .libs/libhello.a foo.o hello.o
ranlib .libs/libhello.a
creating libhello.la
burger$
```

Now that's significantly cooler... libtool just ran an obscure ld command to create a shared library, as well as the static library.

Note how libtool creates extra files in the '.libs' subdirectory, rather than the current directory. This feature is to make it easier to clean up the build directory, and to help ensure that other programs fail horribly if you accidentally forget to use libtool when you should.

3.3 Linking executables

If you choose at this point to *install* the library (put it in a permanent location) before linking executables against it, then you don't need to use libtool to do the linking. Simply use the appropriate '-L' and '-l' flags to specify the library's location.

Some system linkers insist on encoding the full directory name of each shared library in the resulting executable. Libtool has to work around this misfeature by special magic to ensure that only permanent directory names are put into installed executables.

The importance of this bug must not be overlooked: it won't cause programs to crash in obvious ways. It creates a security hole, and possibly even worse, if you are modifying

[1] If you don't specify an rpath, then libtool builds a libtool convenience archive, not a shared library (see Section 3.7 [Static libraries], page 12).

the library source code after you have installed the package, you will change the behaviour of the installed programs!

So, if you want to link programs against the library before you install it, you must use libtool to do the linking.

Here's the old way of linking against an uninstalled library:

```
burger$ gcc -g -O -o hell.old main.o libhello.a -lm
burger$
```

Libtool's way is almost the same[2] (see Section 4.2 [Link mode], page 15):

```
a23$ libtool gcc -g -O -o hell main.o libhello.la -lm
gcc -g -O -o hell main.o ./.libs/libhello.a -lm
a23$
```

That looks too simple to be true. All libtool did was transform '`libhello.la`' to '`./.libs/libhello.a`', but remember that '`a23`' has no shared libraries.

On '`burger`' the situation is different:

```
burger$ libtool gcc -g -O -o hell main.o libhello.la -lm
gcc -g -O -o .libs/hell main.o -L./.libs -R/usr/local/lib -lhello -lm
creating hell
burger$
```

Now assume '`libhello.la`' had already been installed, and you want to link a new program with it. You could figure out where it lives by yourself, then run:

```
burger$ gcc -g -O -o test test.o -L/usr/local/lib -lhello
```

However, unless '`/usr/local/lib`' is in the standard library search path, you won't be able to run `test`. However, if you use libtool to link the already-installed libtool library, it will do The Right Thing (TM) for you:

```
burger$ libtool gcc -g -O -o test test.o /usr/local/lib/libhello.la
gcc -g -O -o .libs/test test.o -Wl,--rpath
-Wl,/usr/local/lib /usr/local/lib/libhello.a -lm
creating test
burger$
```

Note that libtool added the necessary run-time path flag, as well as '`-lm`', the library libhello.la depended upon. Nice, huh?

Since libtool created a wrapper script, you should use libtool to install it and debug it too. However, since the program does not depend on any uninstalled libtool library, it is probably usable even without the wrapper script. Libtool could probably be made smarter to avoid the creation of the wrapper script in this case, but this is left as an exercise for the reader.

Notice that the executable, `hell`, was actually created in the '`.libs`' subdirectory. Then, a wrapper script was created in the current directory.

On NetBSD 1.2, libtool encodes the installation directory of '`libhello`', by using the '`-R/usr/local/lib`' compiler flag. Then, the wrapper script guarantees that the executable finds the correct shared library (the one in '`./.libs`') until it is properly installed.

---

[2] However, you should avoid using '`-L`' or '`-l`' flags to link against an uninstalled libtool library. Just specify the relative path to the '`.la`' file, such as '`../intl/libintl.la`'. This is a design decision to eliminate any ambiguity when linking against uninstalled shared libraries.

Let's compare the two different programs:

```
burger$ time ./hell.old
Welcome to GNU Hell!
** This is not GNU Hello.  There is no built-in mail reader. **
        0.21 real         0.02 user         0.08 sys
burger$ time ./hell
Welcome to GNU Hell!
** This is not GNU Hello.  There is no built-in mail reader. **
        0.63 real         0.09 user         0.59 sys
burger$
```

The wrapper script takes significantly longer to execute, but at least the results are correct, even though the shared library hasn't been installed yet.

So, what about all the space savings that shared libraries are supposed to yield?

```
burger$ ls -l hell.old libhello.a
-rwxr-xr-x  1 gord  gord  15481 Nov 14 12:11 hell.old
-rw-r--r--  1 gord  gord   4274 Nov 13 18:02 libhello.a
burger$ ls -l .libs/hell .libs/libhello.*
-rwxr-xr-x  1 gord  gord  11647 Nov 14 12:10 .libs/hell
-rw-r--r--  1 gord  gord   4274 Nov 13 18:44 .libs/libhello.a
-rwxr-xr-x  1 gord  gord  12205 Nov 13 18:44 .libs/libhello.so.0.0
burger$
```

Well, that sucks. Maybe I should just scrap this project and take up basket weaving.

Actually, it just proves an important point: shared libraries incur overhead because of their (relative) complexity. In this situation, the price of being dynamic is eight kilobytes, and the payoff is about four kilobytes. So, having a shared '`libhello`' won't be an advantage until we link it against at least a few more programs.

3.4 Debugging executables

If '`hell`' was a complicated program, you would certainly want to test and debug it before installing it on your system. In the above section, you saw how the libtool wrapper script makes it possible to run the program directly, but unfortunately, this mechanism interferes with the debugger:

```
burger$ gdb hell
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i386-unknown-netbsd), (C) 1996 Free Software Foundation, Inc.

"hell": not in executable format: File format not recognized

(gdb) quit
burger$
```

Sad. It doesn't work because GDB doesn't know where the executable lives. So, let's try again, by invoking GDB directly on the executable:

```
burger$ gdb .libs/hell
```

```
trick:/home/src/libtool/demo$ gdb .libs/hell
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i386-unknown-netbsd), (C) 1996 Free Software Foundation, Inc.
(gdb) break main
Breakpoint 1 at 0x8048547: file main.c, line 29.
(gdb) run
Starting program: /home/src/libtool/demo/.libs/hell
/home/src/libtool/demo/.libs/hell: can't load library 'libhello.so.2'

Program exited with code 020.
(gdb) quit
burger$
```

Argh. Now GDB complains because it cannot find the shared library that '`hell`' is linked against. So, we must use libtool in order to properly set the library path and run the debugger. Fortunately, we can forget all about the '`.libs`' directory, and just run it on the executable wrapper (see Section 4.3 [Execute mode], page 17):

```
burger$ libtool gdb hell
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is no warranty for GDB; type "show warranty" for details.
GDB 4.16 (i386-unknown-netbsd), (C) 1996 Free Software Foundation, Inc.
(gdb) break main
Breakpoint 1 at 0x8048547: file main.c, line 29.
(gdb) run
Starting program: /home/src/libtool/demo/.libs/hell

Breakpoint 1, main (argc=1, argv=0xbffffc40) at main.c:29
29   printf ("Welcome to GNU Hell!\n");
(gdb) quit
The program is running.  Quit anyway (and kill it)? (y or n) y
burger$
```

3.5 Installing libraries

Installing libraries on a non-libtool system is quite straightforward... just copy them into place:[3]

```
burger$ su
Password: ********
burger# cp libhello.a /usr/local/lib/libhello.a
burger#
```

Oops, don't forget the `ranlib` command:

```
burger# ranlib /usr/local/lib/libhello.a
burger#
```

[3] Don't accidentally strip the libraries, though, or they will be unusable.

Libtool installation is quite simple, as well. Just use the `install` or `cp` command that you normally would (see Section 4.4 [Install mode], page 18):

```
a23# libtool cp libhello.la /usr/local/lib/libhello.la
cp libhello.la /usr/local/lib/libhello.la
cp .libs/libhello.a /usr/local/lib/libhello.a
ranlib /usr/local/lib/libhello.a
a23#
```

Note that the libtool library '`libhello.la`' is also installed, to help libtool with uninstallation (see Section 4.6 [Uninstall mode], page 18) and linking (see Section 3.3 [Linking executables], page 7) and to help programs with dlopening (see Chapter 9 [Dlopened modules], page 34).

Here is the shared library example:

```
burger# libtool install -c libhello.la /usr/local/lib/libhello.la
install -c .libs/libhello.so.0.0 /usr/local/lib/libhello.so.0.0
install -c libhello.la /usr/local/lib/libhello.la
install -c .libs/libhello.a /usr/local/lib/libhello.a
ranlib /usr/local/lib/libhello.a
burger#
```

It is safe to specify the '`-s`' (strip symbols) flag if you use a BSD-compatible install program when installing libraries. Libtool will either ignore the '`-s`' flag, or will run a program that will strip only debugging and compiler symbols from the library.

Once the libraries have been put in place, there may be some additional configuration that you need to do before using them. First, you must make sure that where the library is installed actually agrees with the '`-rpath`' flag you used to build it.

Then, running '`libtool -n --finish` *libdir*' can give you further hints on what to do (see Section 4.5 [Finish mode], page 18):

```
burger# libtool -n --finish /usr/local/lib
PATH="$PATH:/sbin" ldconfig -m /usr/local/lib
-----------------------------------------------------------------
Libraries have been installed in:
   /usr/local/lib

To link against installed libraries in a given directory, LIBDIR,
you must use the `-LLIBDIR' flag during linking.

 You will also need to do one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-RLIBDIR' linker flag

See any operating system documentation about shared libraries for
more information, such as the ld and ld.so manual pages.
-----------------------------------------------------------------
burger#
```

After you have completed these steps, you can go on to begin using the installed libraries. You may also install any executables that depend on libraries you created.

3.6 Installing executables

If you used libtool to link any executables against uninstalled libtool libraries (see Section 3.3 [Linking executables], page 7), you need to use libtool to install the executables after the libraries have been installed (see Section 3.5 [Installing libraries], page 10).

So, for our Ultrix example, we would run:

```
a23# libtool install -c hell /usr/local/bin/hell
install -c hell /usr/local/bin/hell
a23#
```

On shared library systems, libtool just ignores the wrapper script and installs the correct binary:

```
burger# libtool install -c hell /usr/local/bin/hell
install -c .libs/hell /usr/local/bin/hell
burger#
```

3.7 Linking static libraries

Why return to `ar` and `ranlib` silliness when you've had a taste of libtool? Well, sometimes it is desirable to create a static archive that can never be shared. The most frequent case is when you have a set of object files that you use to build several different programs. You can create a "convenience library" out of those objects, and link programs with the library, instead of listing all object files for every program. This technique is often used to overcome GNU automake's lack of support for linking object files built from sources in other directories, because it supports linking with libraries from other directories. This limitation applies to GNU automake up to release 1.4; newer releases should support sources in other directories.

If you just want to link this convenience library into programs, then you could just ignore libtool entirely, and use the old `ar` and `ranlib` commands (or the corresponding GNU automake '`_LIBRARIES`' rules). You can even install a convenience library (but you probably don't want to) using libtool:

```
burger$ libtool ./install-sh -c libhello.a /local/lib/libhello.a
./install-sh -c libhello.a /local/lib/libhello.a
ranlib /local/lib/libhello.a
burger$
```

Using libtool for static library installation protects your library from being accidentally stripped (if the installer used the '`-s`' flag), as well as automatically running the correct `ranlib` command.

But libtool libraries are more than just collections of object files: they can also carry library dependency information, which old archives do not. If you want to create a libtool static convenience library, you can omit the '`-rpath`' flag and use '`-static`' to indicate that you're only interested in a static library. When you link a program with such a library, libtool will actually link all object files and dependency libraries into the program.

If you omit both '-rpath' and '-static', libtool will create a convenience library that can be used to create other libtool libraries, even shared ones. Just like in the static case, the library behaves as an alias to a set of object files and dependency libraries, but in this case the object files are suitable for inclusion in shared libraries. But be careful not to link a single convenience library, directly or indirectly, into a single program or library, otherwise you may get errors about symbol redefinitions.

When GNU automake is used, you should use `noinst_LTLIBRARIES` instead of `lib_LTLIBRARIES` for convenience libraries, so that the '-rpath' option is not passed when they are linked.

As a rule of thumb, link a libtool convenience library into at most one libtool library, and never into a program, and link libtool static convenience libraries only into programs, and only if you need to carry library dependency information to the user of the static convenience library.

Another common situation where static linking is desirable is in creating a standalone binary. Use libtool to do the linking and add the '-all-static' flag.

4 Invoking libtool

The libtool program has the following synopsis:

```
libtool [option]... [mode-arg]...
```

and accepts the following options:

'--config'
: Display libtool configuration variables and exit.

'--debug'
: Dump a trace of shell script execution to standard output. This produces a lot of output, so you may wish to pipe it to less (or more) or redirect to a file.

'-n'
'--dry-run'
: Don't create, modify, or delete any files, just show what commands would be executed by libtool.

'--features'
: Display basic configuration options. This provides a way for packages to determine whether shared or static libraries will be built.

'--finish'
: Same as '--mode=finish'.

'--help'
: Display a help message and exit. If '--mode=*mode*' is specified, then detailed help for *mode* is displayed.

'--mode=*mode*'
: Use *mode* as the operation mode. By default, the operation mode is inferred from the *mode-args*.

  If *mode* is specified, it must be one of the following:

  'compile'
  : Compile a source file into a libtool object.

  'execute'
  : Automatically set the library path so that another program can use uninstalled libtool-generated programs or libraries.

  'finish'
  : Complete the installation of libtool libraries on the system.

  'install'
  : Install libraries or executables.

  'link'
  : Create a library or an executable.

  'uninstall'
  : Delete installed libraries or executables.

  'clean'
  : Delete uninstalled libraries or executables.

'--version'
: Print libtool version information and exit.

The *mode-args* are a variable number of arguments, depending on the selected operation mode. In general, each *mode-arg* is interpreted by programs libtool invokes, rather than libtool itself.

4.1 Compile mode

For *compile* mode, *mode-args* is a compiler command to be used in creating a 'standard' object file. These arguments should begin with the name of the C compiler, and contain the '`-c`' compiler flag so that only an object file is created.

Libtool determines the name of the output file by removing the directory component from the source file name, then substituting the source code suffix (e.g. '`.c`' for C source code) with the library object suffix, '`.lo`'.

If shared libraries are being built, any necessary PIC generation flags are substituted into the compilation command. You can pass compiler and linker specific flags using '`-Wc,`*flag*' and '`-Xcompiler` *flag*' or '`-Wl,`*flag*' and '`-Xlinker` *flag*', respectively.

If the '`-static`' option is given, then a '`.o`' file is built, even if libtool was configured with '`--disable-static`'.

Note that the '`-o`' option is now fully supported. It is emulated on the platforms that don't support it (by locking and moving the objects), so it is really easy to use libtool, just with minor modifications to your Makefiles. Typing for example

```
libtool gcc -c foo/x.c -o foo/x.lo
```

will do what you expect.

Note, however, that, if the compiler does not support '`-c`' and '`-o`', it is impossible to compile '`foo/x.c`' without overwriting an existing '`./x.o`'. Therefore, if you do have a source file '`./x.c`', make sure you introduce dependencies in your '`Makefile`' to make sure '`./x.o`' (or '`./x.lo`') is re-created after any sub-directory's '`x.lo`':

```
x.o x.lo: foo/x.lo bar/x.lo
```

This will also ensure that make won't try to use a temporarily corrupted '`x.o`' to create a program or library. It may cause needless recompilation on platforms that support '`-c`' and '`-o`' together, but it's the only way to make it safe for those that don't.

4.2 Link mode

Link mode links together object files (including library objects) to form another library or to create an executable program.

mode-args consist of a command using the C compiler to create an output file (with the '`-o`' flag) from several object files.

The following components of *mode-args* are treated specially:

'`-all-static`'
: If *output-file* is a program, then do not link it against any shared libraries at all. If *output-file* is a library, then only create a static library.

'`-avoid-version`'
: Tries to avoid versioning (see Chapter 6 [Versioning], page 27) for libraries and modules, i.e. no version information is stored and no symbolic links are created. If the platform requires versioning, this option has no effect.

'`-dlopen` *file*'
: Same as '`-dlpreopen` *file*', if native dlopening is not supported on the host platform (see Chapter 9 [Dlopened modules], page 34) or if the program is linked with '`-static`' or '`-all-static`'. Otherwise, no effect. If *file* is `self` libtool will make sure that the program can `dlopen` itself, either by enabling `-export-dynamic` or by falling back to '`-dlpreopen self`'.

'`-dlpreopen` *file*'
: Link *file* into the output program, and add its symbols to *lt_preloaded_symbols* (see Section 9.2 [Dlpreopening], page 35). If *file* is `self`, the symbols of the program itself will be added to *lt_preloaded_symbols*. If *file* is `force` libtool will make sure that *lt_preloaded_symbols* is always *defined*, regardless of whether it's empty or not.

'`-export-dynamic`'
: Allow symbols from *output-file* to be resolved with `dlsym` (see Chapter 9 [Dlopened modules], page 34).

'`-export-symbols` *symfile*'
: Tells the linker to export only the symbols listed in *symfile*. The symbol file should end in '`.sym`' and must contain the name of one symbol per line. This option has no effect on some platforms. By default all symbols are exported.

'`-export-symbols-regex` *regex*'
: Same as '`-export-symbols`', except that only symbols matching the regular expression *regex* are exported. By default all symbols are exported.

'`-L`*libdir*' Search *libdir* for required libraries that have already been installed.

'`-l`*name*' *output-file* requires the installed library '`lib`*name*'. This option is required even when *output-file* is not an executable.

'`-module`' Creates a library that can be dlopened (see Chapter 9 [Dlopened modules], page 34). This option doesn't work for programs. Module names don't need to be prefixed with 'lib'. In order to prevent name clashes, however, 'libname' and 'name' must not be used at the same time in your package.

'`-no-fast-install`'
: Disable fast-install mode for the executable *output-file*. Useful if the program won't be necessarily installed.

'`-no-install`'
: Link an executable *output-file* that can't be installed and therefore doesn't need a wrapper script. Useful if the program is only used in the build tree, e.g., for testing or generating other files.

'`-no-undefined`'
: Declare that *output-file* does not depend on any other libraries. Some platforms cannot create shared libraries that depend on other libraries (see Chapter 8 [Inter-library dependencies], page 33).

'`-o` *output-file*'
: Create *output-file* from the specified objects and libraries.

'`-release` *release*'
: Specify that the library was generated by release *release* of your package, so that users can easily tell which versions are newer than others. Be warned that no two releases of your package will be binary compatible if you use this flag. If you want binary compatibility, use the '`-version-info`' flag instead (see Chapter 6 [Versioning], page 27).

'`-rpath` *libdir*'
: If *output-file* is a library, it will eventually be installed in *libdir*. If *output-file* is a program, add *libdir* to the run-time path of the program.

'`-R` *libdir*'
: If *output-file* is a program, add *libdir* to its run-time path. If *output-file* is a library, add -R*libdir* to its *dependency_libs*, so that, whenever the library is linked into a program, *libdir* will be added to its run-time path.

'`-static`'
: If *output-file* is a program, then do not link it against any uninstalled shared libtool libraries. If *output-file* is a library, then only create a static library.

'`-version-info` *current*`[:`*revision*`[:`*age*`]]`'
: If *output-file* is a libtool library, use interface version information *current*, *revision*, and *age* to build it (see Chapter 6 [Versioning], page 27). Do **not** use this flag to specify package release information, rather see the '`-release`' flag.

'`-Wl,`*flag*'
'`-Xlinker` *flag*'
: Pass a linker specific flag directly to the linker.

If the *output-file* ends in '`.la`', then a libtool library is created, which must be built only from library objects ('`.lo`' files). The '`-rpath`' option is required. In the current implementation, libtool libraries may not depend on other uninstalled libtool libraries (see Chapter 8 [Inter-library dependencies], page 33).

If the *output-file* ends in '`.a`', then a standard library is created using `ar` and possibly `ranlib`.

If *output-file* ends in '`.o`' or '`.lo`', then a reloadable object file is created from the input files (generally using '`ld -r`'). This method is often called *partial linking*.

Otherwise, an executable program is created.

4.3 Execute mode

For *execute* mode, the library path is automatically set, then a program is executed.

The first of the *mode-args* is treated as a program name, with the rest as arguments to that program.

The following components of *mode-args* are treated specially:

'`-dlopen` *file*'
: Add the directory containing *file* to the library path.

This mode sets the library path environment variable according to any '`-dlopen`' flags.

If any of the *args* are libtool executable wrappers, then they are translated into the name of their corresponding uninstalled binary, and any of their required library directories are added to the library path.

4.4 Install mode

In *install* mode, libtool interprets *mode-args* as an installation command beginning with `cp`, or a BSD-compatible `install` program.

The rest of the *mode-args* are interpreted as arguments to that command.

The command is run, and any necessary unprivileged post-installation commands are also completed.

4.5 Finish mode

Finish mode helps system administrators install libtool libraries so that they can be located and linked into user programs.

Each *mode-arg* is interpreted as the name of a library directory. Running this command may require superuser privileges, so the '`--dry-run`' option may be useful.

4.6 Uninstall mode

Uninstall mode deletes installed libraries, executables and objects.

The first *mode-arg* is the name of the program to use to delete files (typically '`/bin/rm`').

The remaining *mode-args* are either flags for the deletion program (beginning with a '-'), or the names of files to delete.

4.7 Clean mode

Clean mode deletes uninstalled libraries, executables, objects and libtool's temporary files associated with them.

The first *mode-arg* is the name of the program to use to delete files (typically '`/bin/rm`').

The remaining *mode-args* are either flags for the deletion program (beginning with a '-'), or the names of files to delete.

5 Integrating libtool with your package

This chapter describes how to integrate libtool with your packages so that your users can install hassle-free shared libraries.

5.1 Writing 'Makefile' rules for libtool

Libtool is fully integrated with Automake (see section "Introduction" in *The Automake Manual*), starting with Automake version 1.2.

If you want to use libtool in a regular '`Makefile`' (or '`Makefile.in`'), you are on your own. If you're not using Automake 1.2, and you don't know how to incorporate libtool into your package you need to do one of the following:

1. Download Automake (version 1.2 or later) from your nearest GNU mirror, install it, and start using it.
2. Learn how to write '`Makefile`' rules by hand. They're sometimes complex, but if you're clever enough to write rules for compiling your old libraries, then you should be able to figure out new rules for libtool libraries (hint: examine the '`Makefile.in`' in the '`demo`' subdirectory of the libtool distribution... note especially that it was automatically generated from the '`Makefile.am`' by Automake).

5.2 Using Automake with libtool

Libtool library support is implemented under the '`LTLIBRARIES`' primary.

Here are some samples from the Automake '`Makefile.am`' in the libtool distribution's '`demo`' subdirectory.

First, to link a program against a libtool library, just use the '`program_LDADD`' variable:

```
bin_PROGRAMS = hell hell.debug

# Build hell from main.c and libhello.la
hell_SOURCES = main.c
hell_LDADD = libhello.la

# Create an easier-to-debug version of hell.
hell_debug_SOURCES = main.c
hell_debug_LDADD = libhello.la
hell_debug_LDFLAGS = -static
```

The flags '`-dlopen`' or '`-dlpreopen`' (see Section 4.2 [Link mode], page 15) would fit better in the *program_LDADD* variable. Unfortunately, GNU automake, up to release 1.4, doesn't accept these flags in a *program_LDADD* variable, so you have the following alternatives:

- add them to *program_LDFLAGS*, and list the libraries in *program_DEPENDENCIES*, then wait for a release of GNU automake that accepts these flags where they belong;
- surround the flags between quotes, but then you must set *program_DEPENDENCIES* too:

```
program_LDADD = "-dlopen" libfoo.la
program_DEPENDENCIES = libfoo.la
```

- set and 'AC_SUBST' variables *DLOPEN* and *DLPREOPEN* in 'configure.in' and use '@DLOPEN@' and '@DLPREOPEN@' as replacements for the explicit flags '-dlopen' and '-dlpreopen' in 'program_LDADD'. Automake will discard 'AC_SUBST'ed variables from dependencies, so it will behave exactly as we expect it to behave when it accepts these flags in 'program_LDADD'. But hey!, this is ugly!

You may use the 'program_LDFLAGS' variable to stuff in any flags you want to pass to libtool while linking 'program' (such as '-static' to avoid linking uninstalled shared libtool libraries).

Building a libtool library is almost as trivial... note the use of 'libhello_la_LDFLAGS' to pass the '-version-info' (see Chapter 6 [Versioning], page 27) option to libtool:

```
# Build a libtool library, libhello.la for installation in libdir.
lib_LTLIBRARIES = libhello.la
libhello_la_SOURCES = hello.c foo.c
libhello_la_LDFLAGS = -version-info 3:12:1
```

The '-rpath' option is passed automatically by Automake (except for libraries listed as noinst_LTLIBRARIES), so you should not specify it.

See section "The Automake Manual" in *The Automake Manual*, for more information.

5.3 Configuring libtool

Libtool requires intimate knowledge of your compiler suite and operating system in order to be able to create shared libraries and link against them properly. When you install the libtool distribution, a system-specific libtool script is installed into your binary directory.

However, when you distribute libtool with your own packages (see Section 5.4 [Distributing], page 24), you do not always know which compiler suite and operating system are used to compile your package.

For this reason, libtool must be *configured* before it can be used. This idea should be familiar to anybody who has used a GNU configure script. configure runs a number of tests for system features, then generates the 'Makefiles' (and possibly a 'config.h' header file), after which you can run make and build the package.

Libtool adds its own tests to your configure script in order to generate a libtool script for the installer's host machine.

5.3.1 The AC_PROG_LIBTOOL macro

If you are using GNU Autoconf (or Automake), you should add a call to AC_PROG_LIBTOOL to your 'configure.in' file. This macro adds many new tests to the configure script so that the generated libtool script will understand the characteristics of the host:

AC_PROG_LIBTOOL Macro
AM_PROG_LIBTOOL Macro

Add support for the '--enable-shared' and '--disable-shared' configure flags.[1] AM_PROG_LIBTOOL was the old name for this macro, and although supported at the moment is deprecated.

By default, this macro turns on shared libraries if they are available, and also enables static libraries if they don't conflict with the shared libraries. You can modify these defaults by calling either the AC_DISABLE_SHARED or AC_DISABLE_STATIC macros:

```
# Turn off shared libraries during beta-testing, since they
# make the build process take too long.
AC_DISABLE_SHARED
AC_PROG_LIBTOOL
```

The user may specify modified forms of the configure flags '--enable-shared' and '--enable-static' to choose whether shared or static libraries are built based on the name of the package. For example, to have shared 'bfd' and 'gdb' libraries built, but not shared 'libg++', you can run all three configure scripts as follows:

```
trick$ ./configure --enable-shared=bfd,gdb
```

In general, specifying '--enable-shared=*pkgs*' is the same as configuring with '--enable-shared' every package named in the comma-separated *pkgs* list, and every other package with '--disable-shared'. The '--enable-static=*pkgs*' flag behaves similarly, but it uses '--enable-static' and '--disable-static'. The same applies to the '--enable-fast-install=*pkgs*' flag, which uses '--enable-fast-install' and '--disable-fast-install'.

The package name 'default' matches any packages which have not set their name in the PACKAGE environment variable.

This macro also sets the shell variable *LIBTOOL_DEPS*, that you can use to automatically update the libtool script if it becomes out-of-date. In order to do that, add to your 'configure.in':

```
AC_PROG_LIBTOOL
AC_SUBST(LIBTOOL_DEPS)
```

and, to 'Makefile.in' or 'Makefile.am':

```
LIBTOOL_DEPS = @LIBTOOL_DEPS@
libtool: $(LIBTOOL_DEPS)
        $(SHELL) ./config.status --recheck
```

If you are using GNU automake, you can omit the assignment, as automake will take care of it. You'll obviously have to create some dependency on 'libtool'.

AC_LIBTOOL_DLOPEN Macro

Enable checking for dlopen support. This macro should be used if the package makes use of the '-dlopen' and '-dlpreopen' flags, otherwise libtool will assume that the system does not support dlopening. The macro must be called **before** AC_PROG_LIBTOOL.

---

[1] AC_PROG_LIBTOOL requires that you define the 'Makefile' variable top_builddir in your 'Makefile.in'. Automake does this automatically, but Autoconf users should set it to the relative path to the top of your build directory ('../..', for example).

AC_LIBTOOL_WIN32_DLL Macro

This macro should be used if the package has been ported to build clean dlls on win32 platforms. Usually this means that any library data items are exported with `__declspec(dllexport)` and imported with `__declspec(dllimport)`. If this macro is not used, libtool will assume that the package libraries are not dll clean and will build only static libraries on win32 hosts.

This macro must be called **before** `AC_PROG_LIBTOOL`, and provision must be made to pass '`-no-undefined`' to `libtool` in link mode from the package `Makefile`. Naturally, if you pass '`-no-undefined`', you must ensure that all the library symbols **really are** defined at link time!

AC_DISABLE_FAST_INSTALL Macro

Change the default behaviour for `AC_PROG_LIBTOOL` to disable optimization for fast installation. The user may still override this default, depending on platform support, by specifying '`--enable-fast-install`'.

AC_DISABLE_SHARED Macro
AM_DISABLE_SHARED Macro

Change the default behaviour for `AC_PROG_LIBTOOL` to disable shared libraries. The user may still override this default by specifying '`--enable-shared`'.

AC_DISABLE_STATIC Macro
AM_DISABLE_STATIC Macro

Change the default behaviour for `AC_PROG_LIBTOOL` to disable static libraries. The user may still override this default by specifying '`--enable-static`'.

The tests in `AC_PROG_LIBTOOL` also recognize the following environment variables:

CC Variable

The C compiler that will be used by the generated `libtool`. If this is not set, `AC_PROG_LIBTOOL` will look for `gcc` or `cc`.

CFLAGS Variable

Compiler flags used to generate standard object files. If this is not set, `AC_PROG_LIBTOOL` will not use any such flags. It affects only the way `AC_PROG_LIBTOOL` runs tests, not the produced `libtool`.

CPPFLAGS Variable

C preprocessor flags. If this is not set, `AC_PROG_LIBTOOL` will not use any such flags. It affects only the way `AC_PROG_LIBTOOL` runs tests, not the produced `libtool`.

LD Variable

The system linker to use (if the generated `libtool` requires one). If this is not set, `AC_PROG_LIBTOOL` will try to find out what is the linker used by *CC*.

LDFLAGS Variable

The flags to be used by libtool when it links a program. If this is not set, AC_PROG_LIBTOOL will not use any such flags. It affects only the way AC_PROG_LIBTOOL runs tests, not the produced libtool.

LIBS Variable

The libraries to be used by AC_PROG_LIBTOOL when it links a program. If this is not set, AC_PROG_LIBTOOL will not use any such flags. It affects only the way AC_PROG_LIBTOOL runs tests, not the produced libtool.

NM Variable

Program to use rather than checking for nm.

RANLIB Variable

Program to use rather than checking for ranlib.

LN_S Variable

A command that creates a link of a program, a soft-link if possible, a hard-link otherwise. AC_PROG_LIBTOOL will check for a suitable program if this variable is not set.

DLLTOOL Variable

Program to use rather than checking for dlltool. Only meaningful for Cygwin/MS-Windows.

OBJDUMP Variable

Program to use rather than checking for objdump. Only meaningful for Cygwin/MS-Windows.

AS Variable

Program to use rather than checking for as. Only used on Cygwin/MS-Windows at the moment.

When you invoke the libtoolize program (see Section 5.4.1 [Invoking libtoolize], page 24), it will tell you where to find a definition of AC_PROG_LIBTOOL. If you use Automake, the aclocal program will automatically add AC_PROG_LIBTOOL support to your configure script.

Nevertheless, it is advisable to include a copy of 'libtool.m4' in 'acinclude.m4', so that, even if 'aclocal.m4' and 'configure' are rebuilt for any reason, the appropriate libtool macros will be used. The alternative is to hope the user will have a compatible version of 'libtool.m4' installed and accessible for aclocal. This may lead to weird errors when versions don't match.

5.4 Including libtool in your package

In order to use libtool, you need to include the following files with your package:

'`config.guess`'
: Attempt to guess a canonical system name.

'`config.sub`'
: Canonical system name validation subroutine script.

'`ltmain.sh`'
: A generic script implementing basic libtool functionality.

Note that the libtool script itself should *not* be included with your package. See Section 5.3 [Configuring], page 20.

You should use the `libtoolize` program, rather than manually copying these files into your package.

5.4.1 Invoking `libtoolize`

The `libtoolize` program provides a standard way to add libtool support to your package. In the future, it may implement better usage checking, or other features to make libtool even easier to use.

The `libtoolize` program has the following synopsis:

`libtoolize [`*option*`]...`

and accepts the following options:

'`--automake`'
: Work silently, and assume that Automake libtool support is used.

  '`libtoolize --automake`' is used by Automake to add libtool files to your package, when `AC_PROG_LIBTOOL` appears in your '`configure.in`'.

'`--copy`'
'`-c`'
: Copy files from the libtool data directory rather than creating symlinks.

'`--debug`'
: Dump a trace of shell script execution to standard output. This produces a lot of output, so you may wish to pipe it to `less` (or `more`) or redirect to a file.

'`--dry-run`'
'`-n`'
: Don't run any commands that modify the file system, just print them out.

'`--force`'
'`-f`'
: Replace existing libtool files. By default, `libtoolize` won't overwrite existing files.

'`--help`'
: Display a help message and exit.

'`--ltdl`'
: Install libltdl in a subdirectory of your package.

'`--ltdl-tar`'
: Add the file libltdl.tar.gz to your package.

'`--version`'
: Print `libtoolize` version information and exit.

If `libtoolize` detects an explicit call to `AC_CONFIG_AUX_DIR` (see section "The Autoconf Manual" in *The Autoconf Manual*) in your '`configure.in`', it will put the files in the specified directory.

`libtoolize` displays hints for adding libtool support to your package, as well.

5.4.2 Autoconf '.o' macros

The Autoconf package comes with a few macros that run tests, then set a variable corresponding to the name of an object file. Sometimes it is necessary to use corresponding names for libtool objects.

Here are the names of variables that list libtool objects:

LTALLOCA Variable

Substituted by `AC_FUNC_ALLOCA` (see section "The Autoconf Manual" in *The Autoconf Manual*). Is either empty, or contains '`alloca.lo`'.

LTLIBOBJS Variable

Substituted by `AC_REPLACE_FUNCS` (see section "The Autoconf Manual" in *The Autoconf Manual*), and a few other functions.

Unfortunately, the stable release of Autoconf (2.13, at the time of this writing) does not have any way for libtool to provide support for these variables. So, if you depend on them, use the following code immediately before the call to `AC_OUTPUT` in your '`configure.in`':

```
LTLIBOBJS=`echo "$LIBOBJS" | sed 's/\.[^.]* /.lo /g;s/\.[^.]*$/.lo/'`
AC_SUBST(LTLIBOBJS)
LTALLOCA=`echo "$ALLOCA" | sed 's/\.[^.]* /.lo /g;s/\.[^.]*$/.lo/'`
AC_SUBST(LTALLOCA)
AC_OUTPUT(...)
```

5.5 Static-only libraries

When you are developing a package, it is often worthwhile to configure your package with the '`--disable-shared`' flag, or to override the defaults for `AC_PROG_LIBTOOL` by using the `AC_DISABLE_SHARED` Autoconf macro (see Section 5.3.1 [The `AC_PROG_LIBTOOL` macro], page 20). This prevents libtool from building shared libraries, which has several advantages:

- compilation is twice as fast, which can speed up your development cycle,
- debugging is easier because you don't need to deal with any complexities added by shared libraries, and
- you can see how libtool behaves on static-only platforms.

You may want to put a small note in your package '`README`' to let other developers know that '`--disable-shared`' can save them time. The following example note is taken from the GIMP[2] distribution '`README`':

[2] GNU Image Manipulation Program, for those who haven't taken the plunge. See `http://www.gimp.org/`.

```
The GIMP uses GNU Libtool in order to build shared libraries on a
variety of systems. While this is very nice for making usable
binaries, it can be a pain when trying to debug a program. For that
reason, compilation of shared libraries can be turned off by
specifying the ‘--disable-shared’ option to ‘configure’.
```

6 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.

6.1 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.

6.2 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.

6.3 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 15).

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 6.4 [Release numbers], page 29), but be warned that every release of your package will not be binary compatible with any other release.

6.4 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 which you do not want to use '-version-info'. For the 'libbfd' example, the next release which 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.

7 Tips for interface design

Writing a good library interface takes a lot of practice and thorough understanding of the problem that the library is intended to solve.

If you design a good interface, it won't have to change often, you won't have to keep updating documentation, and users won't have to keep relearning how to use the library.

Here is a brief list of tips for library interface design, which may help you in your exploits:

Plan ahead
: Try to make every interface truly minimal, so that you won't need to delete entry points very often.

Avoid interface changes
: Some people love redesigning and changing entry points just for the heck of it (note: *renaming* a function is considered changing an entry point). Don't be one of those people. If you must redesign an interface, then try to leave compatibility functions behind so that users don't need to rewrite their existing code.

Use opaque data types
: The fewer data type definitions a library user has access to, the better. If possible, design your functions to accept a generic pointer (which you can cast to an internal data type), and provide access functions rather than allowing the library user to directly manipulate the data. That way, you have the freedom to change the data structures without changing the interface.

  This is essentially the same thing as using abstract data types and inheritance in an object-oriented system.

Use header files
: If you are careful to document each of your library's global functions and variables in header files, and include them in your library source files, then the compiler will let you know if you make any interface changes by accident (see Section 7.1 [C header files], page 30).

Use the `static` keyword (or equivalent) whenever possible
: The fewer global functions your library has, the more flexibility you'll have in changing them. Static functions and variables may change forms as often as you like... your users cannot access them, so they aren't interface changes.

7.1 Writing C header files

Writing portable C header files can be difficult, since they may be read by different types of compilers:

C++ compilers
: C++ compilers require that functions be declared with full prototypes, since C++ is more strongly typed than C. C functions and variables also need to be declared with the `extern "C"` directive, so that the names aren't mangled. See Section 11.1 [C++ libraries], page 52, for other issues relevant to using C++ with libtool.

ANSI C compilers
: ANSI C compilers are not as strict as C++ compilers, but functions should be prototyped to avoid unnecessary warnings when the header file is `#included`.

non-ANSI C compilers
: Non-ANSI compilers will report errors if functions are prototyped.

These complications mean that your library interface headers must use some C preprocessor magic in order to be usable by each of the above compilers.

'`foo.h`' in the '`demo`' subdirectory of the libtool distribution serves as an example for how to write a header file that can be safely installed in a system directory.

Here are the relevant portions of that file:

```
/* BEGIN_C_DECLS should be used at the beginning of your declarations,
   so that C++ compilers don't mangle their names.  Use END_C_DECLS at
   the end of C declarations. */
#undef BEGIN_C_DECLS
#undef END_C_DECLS
#ifdef __cplusplus
# define BEGIN_C_DECLS extern "C" {
# define END_C_DECLS }
#else
# define BEGIN_C_DECLS /* empty */
# define END_C_DECLS /* empty */
#endif

/* PARAMS is a macro used to wrap function prototypes, so that
   compilers that don't understand ANSI C prototypes still work,
   and ANSI C compilers can issue warnings about type mismatches. */
#undef PARAMS
#if defined (__STDC__) || defined (_AIX) \
        || (defined (__mips) && defined (_SYSTYPE_SVR4)) \
        || defined(WIN32) || defined(__cplusplus)
# define PARAMS(protos) protos
#else
# define PARAMS(protos) ()
#endif
```

These macros are used in '`foo.h`' as follows:

```
#ifndef FOO_H
#define FOO_H 1

/* The above macro definitions. */
#include "..."

BEGIN_C_DECLS

int foo PARAMS((void));
int hello PARAMS((void));

END_C_DECLS
```

```
#endif /* !FOO_H */
```

Note that the '#ifndef FOO_H' prevents the body of 'foo.h' from being read more than once in a given compilation.

Also the only thing that must go outside the BEGIN_C_DECLS/END_C_DECLS pair are #include lines. Strictly speaking it is only C symbol names that need to be protected, but your header files will be more maintainable if you have a single pair of of these macros around the majority of the header contents.

You should use these definitions of PARAMS, BEGIN_C_DECLS, and END_C_DECLS into your own headers. Then, you may use them to create header files that are valid for C++, ANSI, and non-ANSI compilers[1].

Do not be naive about writing portable code. Following the tips given above will help you miss the most obvious problems, but there are definitely other subtle portability issues. You may need to cope with some of the following issues:

- Pre-ANSI compilers do not always support the void * generic pointer type, and so need to use char * in its place.
- The const, inline and signed keywords are not supported by some compilers, especially pre-ANSI compilers.
- The long double type is not supported by many compilers.

[1] We used to recommend __P, __BEGIN_DECLS and __END_DECLS. This was bad advice since symbols (even preprocessor macro names) that begin with an underscore are reserved for the use of the compiler.

8 Inter-library dependencies

By definition, every shared library system provides a way for executables to depend on libraries, so that symbol resolution is deferred until runtime.

An *inter-library dependency* is one in which a library depends on other libraries. For example, if the libtool library '`libhello`' uses the `cos` function, then it has an inter-library dependency on '`libm`', the math library that implements `cos`.

Some shared library systems provide this feature in an internally-consistent way: these systems allow chains of dependencies of potentially infinite length.

However, most shared library systems are restricted in that they only allow a single level of dependencies. In these systems, programs may depend on shared libraries, but shared libraries may not depend on other shared libraries.

In any event, libtool provides a simple mechanism for you to declare inter-library dependencies: for every library '`lib`*name*' that your own library depends on, simply add a corresponding `-l`*name* option to the link line when you create your library. To make an example of our '`libhello`' that depends on '`libm`':

```
burger$ libtool gcc -g -O -o libhello.la foo.lo hello.lo \
                -rpath /usr/local/lib -lm
burger$
```

When you link a program against '`libhello`', you don't need to specify the same '`-l`' options again: libtool will do that for you, in order to guarantee that all the required libraries are found. This restriction is only necessary to preserve compatibility with static library systems and simple dynamic library systems.

Some platforms, such as AIX, do not even allow you this flexibility. In order to build a shared library, it must be entirely self-contained (that is, have references only to symbols that are found in the '`.lo`' files or the specified '`-l`' libraries), and you need to specify the *-no-undefined* flag. By default, libtool builds only static libraries on these kinds of platforms.

The simple-minded inter-library dependency tracking code of libtool releases prior to 1.2 was disabled because it was not clear when it was possible to link one library with another, and complex failures would occur. A more complex implementation of this concept was re-introduced before release 1.3, but it has not been ported to all platforms that libtool supports. The default, conservative behavior is to avoid linking one library with another, introducing their inter-dependencies only when a program is linked with them.

9 Dlopened modules

It can sometimes be confusing to discuss *dynamic linking*, because the term is used to refer to two different concepts:

1. Compiling and linking a program against a shared library, which is resolved automatically at run time by the dynamic linker. In this process, dynamic linking is transparent to the application.
2. The application calling functions such as `dlopen`,[1] which load arbitrary, user-specified modules at runtime. This type of dynamic linking is explicitly controlled by the application.

To mitigate confusion, this manual refers to the second type of dynamic linking as *dlopening* a module.

The main benefit to dlopening object modules is the ability to access compiled object code to extend your program, rather than using an interpreted language. In fact, dlopen calls are frequently used in language interpreters to provide an efficient way to extend the language.

As of version 1.4.2, libtool provides support for dlopened modules. However, you should indicate that your package is willing to use such support, by using the macro '`AC_LIBTOOL_DLOPEN`' in '`configure.in`'. If this macro is not used (or it is used *after* '`AC_PROG_LIBTOOL`'), libtool will assume no dlopening mechanism is available, and will try to simulate it.

This chapter discusses how you as a dlopen application developer might use libtool to generate dlopen-accessible modules.

9.1 Building modules to dlopen

On some operating systems, a program symbol must be specially declared in order to be dynamically resolved with the `dlsym` (or equivalent) function.

Libtool provides the '`-export-dynamic`' and '`-module`' link flags (see Section 4.2 [Link mode], page 15), which do this declaration. You need to use these flags if you are linking an application program that dlopens other modules or a libtool library that will also be dlopened.

For example, if we wanted to build a shared library, '`libhello`', that would later be dlopened by an application, we would add '`-module`' to the other link flags:

```
burger$ libtool gcc -module -o libhello.la foo.lo \
                hello.lo -rpath /usr/local/lib -lm
burger$
```

If symbols from your *executable* are needed to satisfy unresolved references in a library you want to dlopen you will have to use the flag '`-export-dynamic`'. You should use '`-export-dynamic`' while linking the executable that calls dlopen:

```
burger$ libtool gcc -export-dynamic -o hell-dlopener main.o
burger$
```

---

[1] HP-UX, to be different, uses a function named `shl_load`.

9.2 Dlpreopening

Libtool provides special support for dlopening libtool object and libtool library files, so that their symbols can be resolved *even on platforms without any* `dlopen` *and* `dlsym` *functions*.

Consider the following alternative ways of loading code into your program, in order of increasing "laziness":

1. Linking against object files that become part of the program executable, whether or not they are referenced. If an object file cannot be found, then the linker refuses to create the executable.
2. Declaring a static library to the linker, so that it is searched at link time in order to satisfy any undefined references in the above object files. If the static library cannot be found, then the linker refuses to link the executable.
3. Declaring a shared library to the runtime linker, so that it is searched at runtime in order to satisfy any undefined references in the above files. If the shared library cannot be found, then the dynamic linker aborts the program before it runs.
4. Dlopening a module, so that the application can resolve its own, dynamically-computed references. If there is an error opening the module, or the module is not found, then the application can recover without crashing.

Libtool emulates '`-dlopen`' on static platforms by linking objects into the program at compile time, and creating data structures that represent the program's symbol table.

In order to use this feature, you must declare the objects you want your application to dlopen by using the '`-dlopen`' or '`-dlpreopen`' flags when you link your program (see Section 4.2 [Link mode], page 15).

`struct` **lt_dlsymlist** `{ const char *`*name*`; lt_ptr` *address*`; }` Structure

The *name* attribute is a null-terminated character string of the symbol name, such as `"fprintf"`. The *address* attribute is a generic pointer to the appropriate object, such as `&fprintf`.

`const lt_dlsymlist *` **lt_preloaded_symbols** Variable

An array of *lt_symbol* structures, representing all the preloaded symbols linked into the program. For each '`-dlpreloaded`' file there is an element with the *name* of the file and a *address* of `0`, followed by all symbols exported from this file. For the executable itself the special name @PROGRAM@ is used. The last element has a *name* and *address* of `0`.

Some compilers may allow identifiers which are not valid in ANSI C, such as dollar signs. Libtool only recognizes valid ANSI C symbols (an initial ASCII letter or underscore, followed by zero or more ASCII letters, digits, and underscores), so non-ANSI symbols will not appear in *lt_preloaded_symbols*.

9.3 Finding the correct name to dlopen

After a library has been linked with '-module', it can be dlopened. Unfortunately, because of the variation in library names, your package needs to determine the correct file to dlopen.

The most straightforward and flexible implementation is to determine the name at runtime, by finding the installed '.la' file, and searching it for the following lines:

```
# The name that we can dlopen.
dlname='dlname'
```

If *dlname* is empty, then the library cannot be dlopened. Otherwise, it gives the dlname of the library. So, if the library was installed as '/usr/local/lib/libhello.la', and the *dlname* was 'libhello.so.3', then '/usr/local/lib/libhello.so.3' should be dlopened.

If your program uses this approach, then it should search the directories listed in the LD_LIBRARY_PATH[2] environment variable, as well as the directory where libraries will eventually be installed. Searching this variable (or equivalent) will guarantee that your program can find its dlopened modules, even before installation, provided you have linked them using libtool.

9.4 Unresolved dlopen issues

The following problems are not solved by using libtool's dlopen support:

- Dlopen functions are generally only available on shared library platforms. If you want your package to be portable to static platforms, you have to use either libltdl (see Chapter 10 [Using libltdl], page 37) or develop your own alternatives to dlopening dynamic code. Most reasonable solutions involve writing wrapper functions for the dlopen family, which do package-specific tricks when dlopening is unsupported or not available on a given platform.
- There are major differences in implementations of the dlopen family of functions. Some platforms do not even use the same function names (notably HP-UX, with its shl_load family).
- The application developer must write a custom search function in order to discover the correct module filename to supply to dlopen.

[2] LIBPATH on AIX, and SHLIB_PATH on HP-UX.

10 Using libltdl

Libtool provides a small library, called 'libltdl', that aims at hiding the various difficulties of dlopening libraries from programmers. It consists of a header-file and a small C source file that can be distributed with applications that need dlopening functionality. On some platforms, whose dynamic linkers are too limited for a simple implementation of 'libltdl' services, it requires GNU DLD, or it will only emulate dynamic linking with libtool's dlpreopening mechanism.

libltdl supports currently the following dynamic linking mechanisms:

- `dlopen` (Solaris, Linux and various BSD flavors)
- `shl_load` (HP-UX)
- `LoadLibrary` (Win16 and Win32)
- `load_add_on` (BeOS)
- GNU DLD (emulates dynamic linking for static libraries)
- libtool's dlpreopen (see see Section 9.2 [Dlpreopening], page 35)

libltdl is licensed under the terms of the GNU Library General Public License, with the following exception:

> As a special exception to the GNU Lesser General Public License, if you distribute this file as part of a program or library that is built using GNU libtool, you may include it under the same distribution terms that you use for the rest of that program.

10.1 How to use libltdl in your programs

The libltdl API is similar to the dlopen interface of Solaris and Linux, which is very simple but powerful.

To use libltdl in your program you have to include the header file 'ltdl.h':

```
#include <ltdl.h>
```

The last release of libltdl used some symbols that violated the POSIX namespace conventions. These symbols are now deprecated, and have been replaced by those described here. If you have code that relies on the old deprecated symbol names, defining 'LT_NON_POSIX_NAMESPACE' before you include 'ltdl.h' provides conversion macros. Whichever set of symbols you use, the new api is not binary compatible with the last, so you will need to recompile your application in order to use this version of libltdl.

Note that libltdl is not threadsafe, i.e. a multithreaded application has to use a mutex for libltdl. It was reported that GNU/Linux's glibc 2.0's `dlopen` with 'RTLD_LAZY' (which libltdl uses by default) is not thread-safe, but this problem is supposed to be fixed in glibc 2.1. On the other hand, 'RTLD_NOW' was reported to introduce problems in multi-threaded applications on FreeBSD. Working around these problems is left as an exception for the reader; contributions are certainly welcome.

The following types are defined in 'ltdl.h':

lt_ptr Type

`lt_ptr` is a generic pointer.

lt_dlhandle Type

`lt_dlhandle` is a module "handle". Every lt_dlopened module has a handle associated with it.

lt_dlsymlist Type

`lt_dlsymlist` is a symbol list for dlpreopened modules. This structure is described in see Section 9.2 [Dlpreopening], page 35.

libltdl provides the following functions:

`int` **lt_dlinit** `(void)` Function

Initialize libltdl. This function must be called before using libltdl and may be called several times. Return 0 on success, otherwise the number of errors.

`int` **lt_dlexit** `(void)` Function

Shut down libltdl and close all modules. This function will only then shut down libltdl when it was called as many times as `lt_dlinit` has been successfully called. Return 0 on success, otherwise the number of errors.

`lt_dlhandle` **lt_dlopen** `(const char *`*filename*`)` Function

Open the module with the file name *filename* and return a handle for it. `lt_dlopen` is able to open libtool dynamic modules, preloaded static modules, the program itself and native dynamic libraries.

Unresolved symbols in the module are resolved using its dependency libraries (not implemented yet) and previously dlopened modules. If the executable using this module was linked with the `-export-dynamic` flag, then the global symbols in the executable will also be used to resolve references in the module.

If *filename* is `NULL` and the program was linked with `-export-dynamic` or `-dlopen self`, `lt_dlopen` will return a handle for the program itself, which can be used to access its symbols.

If libltdl cannot find the library and the file name *filename* does not have a directory component it will additionally search in the following search paths for the module (in the order as follows):

1. user-defined search path: This search path can be set by the program using the functions `lt_dlsetsearchpath` and `lt_dladdsearchdir`.
2. libltdl's search path: This search path is the value of the environment variable *LTDL_LIBRARY_PATH*.
3. system library search path: The system dependent library search path (e.g. on Linux it is *LD_LIBRARY_PATH*).

Each search path must be a colon-separated list of absolute directories, for example, `"/usr/lib/mypkg:/lib/foo"`.

If the same module is loaded several times, the same handle is returned. If `lt_dlopen` fails for any reason, it returns `NULL`.

`lt_dlhandle` **lt_dlopenext** `(const char *`*filename*`)` Function

The same as `lt_dlopen`, except that it tries to append different file name extensions to the file name. If the file with the file name *filename* cannot be found libltdl tries to append the following extensions:

1. the libtool archive extension '`.la`'
2. the extension used for native dynamic libraries on the host platform, e.g., '`.so`', '`.sl`', etc.

This lookup strategy was designed to allow programs that don't have knowledge about native dynamic libraries naming conventions to be able to `dlopen` such libraries as well as libtool modules transparently.

int **lt_dlclose** (`lt_dlhandle` *handle*) — Function

Decrement the reference count on the module *handle*. If it drops to zero and no other module depends on this module, then the module is unloaded. Return 0 on success.

lt_ptr **lt_dlsym** (`lt_dlhandle` *handle*, `const char` **name*) — Function

Return the address in the module *handle*, where the symbol given by the null-terminated string *name* is loaded. If the symbol cannot be found, `NULL` is returned.

const char *lt_dlerror **(void)** — Function

Return a human readable string describing the most recent error that occurred from any of libltdl's functions. Return `NULL` if no errors have occurred since initialization or since it was last called.

int **lt_dlpreload** (`const lt_dlsymlist` **preloaded*) — Function

Register the list of preloaded modules *preloaded*. If *preloaded* is `NULL`, then all previously registered symbol lists, except the list set by `lt_dlpreload_default`, are deleted. Return 0 on success.

int **lt_dlpreload_default** (`const lt_dlsymlist` **preloaded*) — Function

Set the default list of preloaded modules to *preloaded*, which won't be deleted by `lt_dlpreload`. Note that this function does *not* require libltdl to be initialized using `lt_dlinit` and can be used in the program to register the default preloaded modules. Instead of calling this function directly, most programs will use the macro `LTDL_SET_PRELOADED_SYMBOLS`.

Return 0 on success.

LTDL_SET_PRELOADED_SYMBOLS() — Macro

Set the default list of preloaded symbols. Should be used in your program to initialize libltdl's list of preloaded modules.

```
#include <ltdl.h>

int main() {
  /* ... */
  LTDL_SET_PRELOADED_SYMBOLS();
  /* ... */
}
```

int **lt_dladdsearchdir** (`const char` **search_dir*) — Function

Add the search directory *search_dir* to the user-defined library search path. Return 0 on success.

`int` **lt_dlsetsearchpath** (`const char *`*search_path*) Function

Replace the current user-defined library search path with *search_path*, which must be a colon-separated list of absolute directories. Return 0 on success.

`const char *lt_dlgetsearchpath` **(void)** Function

Return the current user-defined library search path.

`int` **lt_dlmakeresident** (`lt_dlhandle` *handle*) Function

Mark a module so that it cannot be '`lt_dlclose`'d. This can be useful if a module implements some core functionality in your project, which would cause your code to crash if removed. Return 0 on success.

If you use '`lt_dlopen (NULL)`' to get a *handle* for the running binary, that handle will always be marked as resident, and consequently cannot be successfully '`lt_dlclose`'d.

`int` **lt_dlisresident** (`lt_dlhandle` *handle*) Function

Check whether a particular module has been marked as resident, returning 1 if it has or 0 otherwise. If there is an error while executing this function, return -1 and set an error message for retrieval with `lt_dlerror`.

`lt_ptr (*) (size_t` *size*`)` **lt_dlmalloc** Variable

`void (*) (lt_ptr` *ptr*`)` **lt_dlfree** Variable

These variables are set to `malloc` and `free`, by default, but you can set them to any other functions that provides equivalent functionality. However, you must not modify their values after calling any libltdl function other than `lt_dlpreopen_default` or the macro `LTDL_SET_PRELOADED_SYMBOLS`.

10.2 Creating modules that can be `dlopen`ed

Libtool modules are like normal libtool libraries with a few exceptions:

You have to link the module with libtool's '`-module`' switch, and you should link any program that is intended to dlopen the module with '`-dlopen modulename.la`' so that libtool can dlpreopen the module on platforms which don't support dlopening. If the module depends on any other libraries, make sure you specify them either when you link the module or when you link programs that dlopen it. If you want to disable see Chapter 6 [Versioning], page 27 for a specific module you should link it with the '`-avoid-version`' switch. Note that libtool modules don't need to have a "lib" prefix. However, automake 1.4 or higher is required to build such modules.

Usually a set of modules provide the same interface, i.e, exports the same symbols, so that a program can dlopen them without having to know more about their internals. In order to avoid symbol conflicts all exported symbols must be prefixed with "modulename_LTX_" ('`modulename`' is the name of the module). Internal symbols must be named in such a way that they won't conflict with other modules, for example, by prefixing them with "_modulename_". Although some platforms support having the same symbols defined more than once it is generally not portable and it makes it impossible to dlpreopen such modules. libltdl will automatically cut the prefix off to get the real name of the symbol. Additionally, it supports modules which don't use a prefix so that you can also dlopen non-libtool modules.

'foo1.c' gives an example of a portable libtool module. Exported symbols are prefixed with "foo1_LTX_", internal symbols with "_foo1_". Aliases are defined at the beginning so that the code is more readable.

```
/* aliases for the exported symbols */
#define foo foo1_LTX_foo
#define bar foo1_LTX_bar

/* a global variable definition */
int bar = 1;

/* a private function */
int _foo1_helper() {
  return bar;
}

/* an exported function */
int foo() {
  return _foo1_helper();
}
```

The 'Makefile.am' contains the necessary rules to build the module 'foo1.la':

```
...
lib_LTLIBRARIES = foo1.la

foo1_la_SOURCES = foo1.c
foo1_la_LDFLAGS = -module
...
```

10.3 Using libtldl in a multi threaded environment

Using the `lt_dlmutex_register()` function, and by providing some appropriate callback function definitions, libltdl can be used in a multi-threaded environment.

`void` **lt_dlmutex_lock** (`void`) Type

This is the type of a function pointer holding the address of a function which will be called at the start of parts of the libltdl implementation code which require a mutex lock.

Because libltdl is inherantly recursive, it is important that the locking mechanism employed by these callback functions are reentrant, or else strange problems will occur.

`void` **lt_dlmutex_unlock** (`void`) Type

The type of a matching unlock function.

`void` **lt_dlmutex_seterror** (`const char *`*error*); Type

Many of the functions in the libltdl API have a special return value to indicate to the client that an error has occured. Normally (in single threaded applications) a string describing that error can be retrieved from internal storage with `lt_dlerror()`.

A function of this type must be registered with the library in order for it to work in a multi-threaded context. The function should store any error message passed in thread local storage.

const char * **lt_dlmutex_geterror** (void) — Type

The type of a matching callback function to retrieve the last stored error message from thread local storage.

When regeistered correctly this function will be used by `lt_dlerror())` from all threads to retrieve error messages for the client.

int **lt_dlmutex_register** (lt_dlmutex_lock **lock*, lt_dlmutex_unlock **unlock*, lt_dlmutex_set_error **seterror*, lt_dlmutex_geterror **geterror*) — Function

Use this function to register one of each of function ttypes described above in preparation for multi-threaded use of libltdl. All arguments must be valid non-`NULL` function addresses, or else all `NULL` to return to single threaded operation.

10.4 Data associated with loaded modules

Some of the internal information about each loaded module that is maintained by libltdl is available to the user, in the form of this structure:

struct **lt_dlinfo** { char **filename*; char **name*; int *ref_count*; } — Type

`lt_dlinfo` is used to store information about a module. The *filename* attribute is a null-terminated character string of the real module file name. If the module is a libtool module then *name* is its module name (e.g. `"libfoo"` for `"dir/libfoo.la"`), otherwise it is set to `NULL`. The *ref_count* attribute is a reference counter that describes how often the same module is currently loaded.

The following function will return a pointer to libltdl's internal copy of this structure for the given *handle*:

const lt_dlinfo * **lt_dlgetinfo** (lt_dlhandle *handle*) — Function

Return a pointer to a struct that contains some information about the module *handle*. The contents of the struct must not be modified. Return `NULL` on failure.

Furthermore, in order to save you from having to keep a list of the handles of all the modules you have loaded, these functions allow you to iterate over libltdl's list of loaded modules:

int **lt_dlforeach** (int (**func*) (lt_dlhandle *handle*, lt_ptr *data*), lt_ptr *data*) — Function

For each loaded module call the function *func*. The argument *handle* is the handle of one of the loaded modules, *data* is the *data* argument passed to `lt_dlforeach`. As soon as *func* returns a non-zero value for one of the handles, `lt_dlforeach` will stop calling *func* and immediately return 1. Otherwise 0 is returned.

`lt_dlhandle` **lt_dlhandle_next** (`lt_dlhandle place`) Function

Iterate over the loaded module handles, returning the first handle in the list if *place* is `NULL`, and the next one on subsequent calls. If *place* is the last element in the list of loaded modules, this function returns `NULL`.

Of course, you would still need to maintain your own list of loaded module handles to parallel the list maintained by libltdl if there are any other data that you need to associate with each handle for the purposes of your application. However, if you use the following API calls to associate your application data with individual module handles as they are loaded there is actually no need to do that. You must first obtain a unique caller id from libltdl which you subsequently use to retrieve the data you stored earlier. This allows for different libraries that each wish to store their own data against loaded modules to do so without interfering with one another's data.

lt_dlcaller_id Type

The opaque type used to hold individual data set keys.

`lt_dlcaller_id` **lt_dlcaller_register** (`void`) Function

Use this to obtain a unique key to store and retrieve individual sets of per module data.

`lt_ptr` **lt_dlcaller_set_data** (`lt_dlcaller_id` *key*, `lt_dlhandle` *handle*, `lt_ptr` *data*) Function

Set *data* as the set of data uniquely associated with *key* and *handle* for later retrieval. This function returns the *data* previously associated with *key* and *handle* if any. A result of 0, may indicate that a diagnostic for the last error (if any) is available from `lt_dlerror()`.

For example, to correctly remove some associated data:

```
lt_ptr stale = lt_dlcaller_set_data (key, handle, 0);
if (stale == NULL)
  {
    char *error_msg = lt_dlerror ();

    if (error_msg != NULL)
      {
        my_error_handler (error_msg);
        return STATUS_FAILED;
      }
  }
else
  {
    free (stale);
  }
```

`lt_ptr` **lt_dlcaller_get_data** (`lt_dlcaller_id` *key*, `lt_dlhandle` *handle*) Function

Return the address of the data associated with *key* and *handle*, or else `NULL` if there is none.

The preceding functions can be combined with `lt_dlforeach` to implement search and apply operations without the need for your application to track the modules that have been loaded and unloaded:

```
int
my_dlcaller_callback (lt_dlhandle handle, lt_ptr key_ptr)
{
  struct my_module_data *my_data;

  my_data = lt_dlcaller_get_data (handle, (lt_dlcaller_id) *key_ptr);

  return process (my_data);
}

int
my_dlcaller_foreach (lt_dlcaller_id key)
{
  lt_dlforeach (my_dlcaller_callback, (lt_ptr) &key);
}
```

10.5 How to create and register new module loaders

Sometimes libltdl's many ways of gaining access to modules are not sufficient for the purposes of a project. You can write your own loader, and register it with libltdl so that `lt_dlopen` will be able to use it.

Writing a loader involves writing at least three functions which can be called by `lt_dlopen`, `lt_dlsym` and `lt_dlclose`. Optionally, you can provide a finalisation function to perform any cleanup operations when `lt_dlexit` executes, and a symbol prefix string which will be prepended to any symbols passed to `lt_dlsym`. These functions must match the function pointer types below, after which they can be allocated to an instance of `lt_user_dlloader` and registered.

Registering the loader requires that you choose a name for it, so that it can be recognised by `lt_dlloader_find` and removed with `lt_dlloader_remove`. The name you choose must be unique, and not already in use by libltdl's builtin loaders:

"dlopen"
: The system dynamic library loader, if one exists.

"dld"
: The GNU dld loader, if '`libdld`' was installed when libltdl was built.

"dlpreload"
: The loader for `lt_dlopen`ing of preloaded static modules.

The prefix "dl" is reserved for loaders supplied with future versions of libltdl, so you should not use that for your own loader names.

The following types are defined in '`ltdl.h`':

lt_module Type

`lt_module` is a dlloader dependent module. The dynamic module loader extensions communicate using these low level types.

lt_dlloader Type

`lt_dlloader` is a handle for module loader types.

lt_dlloader_data Type

`lt_dlloader_data` is used for specifying loader instance data.

`struct` **lt_user_dlloader** `{const char` **sym_prefix*`;` Type
`lt_module_open` **module_open*`;lt_module_close` **module_close*`;`
`lt_find_sym` **find_sym*`; lt_dlloader_exit` **dlloader_exit*`;`
`lt_dlloader_data` *dlloader_data*`; }`

If you want to define a new way to open dynamic modules, and have the `lt_dlopen` API use it, you need to instantiate one of these structures and pass it to `lt_dlloader_add`. You can pass whatever you like in the *dlloader_data* field, and it will be passed back as the value of the first parameter to each of the functions specified in the function pointer fields.

`lt_module` **lt_module_open** `(lt_user_data` *loader_data*`,` Type
`const char` **filename*`)`

The type of the loader function for an `lt_dlloader` module loader. The value set in the dlloader_data field of the `struct lt_user_dlloader` structure will be passed into this function in the *loader_data* parameter. Implementation of such a function should attempt to load the named module, and return an `lt_module` suitable for passing in to the associated `lt_module_close` and `lt_sym_find` function pointers. If the function fails it should return `NULL`, and set the error message with `lt_dlseterror`.

`int` **lt_module_close** `(lt_dlloader_data` *loader_data*`,` Type
`lt_module` *module*`)`

The type of the unloader function for a user defined module loader. Implementatation of such a function should attempt to release any resources tied up by the *module* module, and then unload it from memory. If the function fails for some reason, set the error message with `lt_dlseterror` and return non-zero.

`lt_ptr` **lt_find_sym** `(lt_user_data` *loader_data*`, lt_module` *module*`,` Type
`const char` **symbol*`)`

The type of the symbol lookup function for a user defined module loader. Implementation of such a function should return the address of the named *symbol* in the module *module*, or else set the error message with `lt_dlseterror` and return `NULL` if lookup fails.

`int` **lt_dlloader_exit** `(lt_user_data` *loader_data*`)` Type

The type of the finalisation function for a user defined module loader. Implementation of such a function should free any resources associated with the loader, including any user specified data in the `dlloader_data` field of the `lt_user_dlloader`. If non-`NULL`, the function will be called by `lt_dlexit`, and `lt_dlloader_remove`.

For example:

```
int
register_myloader (void)
{
  lt_user_dlloader dlloader;

  /* User modules are responsible for their own initialisation. */
  if (myloader_init () != 0)
    return MYLOADER_INIT_ERROR;

  dlloader.sym_prefix    = NULL;
  dlloader.module_open   = myloader_open;
  dlloader.module_close  = myloader_close;
  dlloader.find_sym      = myloader_find_sym.
  dlloader.dlloader_exit = myloader_exit;
  dlloader.dlloader_data = (lt_user_data)myloader_function;

  /* Add my loader as the default module loader. */
  if (lt_dlloader_add (lt_dlloader_next (NULL), &dlloader, "myloader") != 0)
    return ERROR;

  return OK;
}
```

Note that if there is any initialisation required for the loader, it must be performed manually before the loader is registered – libltdl doesn't handle user loader initialisation.

Finalisation *is* handled by libltdl however, and it is important to ensure the `dlloader_exit` callback releases any resources claimed during the initialisation phase.

libltdl provides the following functions for writing your own module loaders:

int **lt_dlloader_add** (lt_dlloader **place*, lt_user_dlloader **dlloader*, const char **loader_name*) — Function

Add a new module loader to the list of all loaders, either as the last loader (if *place* is NULL), else immediately before the loader passed as *place*. *loader_name* will be returned by `lt_dlloader_name` if it is subsequently passed a newly registered loader. These *loader_name*s must be unique, or `lt_dlloader_remove` and `lt_dlloader_find` cannot work. Returns 0 for success.

```
{
  /* Make myloader be the last one. */
  if (lt_dlloader_add (NULL, myloader) != 0)
    perror (lt_dlerror ());
}
```

int **lt_dlloader_remove** (const char **loader_name*) — Function

Remove the loader identified by the unique name, *loader_name*. Before this can succeed, all modules opened by the named loader must have been closed. Returns 0 for success, otherwise an error message can be obtained from `lt_dlerror`.

```
{
  /* Remove myloader. */
  if (lt_dlloader_remove ("myloader") != 0)
    perror (lt_dlerror ());
}
```

lt_dlloader *lt_dlloader_next **(lt_dlloader ***place*) — Function

Iterate over the module loaders, returning the first loader if *place* is NULL, and the next one on subsequent calls. The handle is for use with `lt_dlloader_add`.

```
{
  /* Make myloader be the first one. */
  if (lt_dlloader_add (lt_dlloader_next (NULL), myloader) != 0)
    return ERROR;
}
```

lt_dlloader *lt_dlloader_find **(const char ***loader_name*) — Function

Return the first loader with a matching *loader_name* identifier, or else NULL, if the identifier is not found.

The identifiers which may be used by libltdl itself, if the host architecture supports them are *dlopen*[1], *dld* and *dlpreload*.

```
{
  /* Add a user loader as the next module loader to be tried if
     the standard dlopen loader were to fail when lt_dlopening. */
  if (lt_dlloader_add (lt_dlloader_find ("dlopen"), myloader) != 0)
    return ERROR;
}
```

[1] This is used for the host dependent module loading API – `shl_load` and `LoadLibrary` for example

const char *lt_dlloader_name **(lt_dlloader** **place***)** Function

Return the identifying name of *PLACE*, as obtained from `lt_dlloader_next` or `lt_dlloader_find`. If this function fails, it will return `NULL` and set an error for retrieval with `lt_dlerror`.

lt_user_data *lt_dlloader_data **(lt_dlloader** **place***)** Function

Return the address of the `dlloader_data` of *PLACE*, as obtained from `lt_dlloader_next` or `lt_dlloader_find`. If this function fails, it will return `NULL` and set an error for retrieval with `lt_dlerror`.

10.5.1 Error handling within user module loaders

int **lt_dladderror** (const char **diagnostic*) Function

This function allows you to integrate your own error messages into `lt_dlerror`. Pass in a suitable diagnostic message for return by `lt_dlerror`, and an error identifier for use with `lt_dlseterror` is returned.

If the allocation of an identifier fails, this function returns -1.

```
int myerror = lt_dladderror ("Doh!");
if (myerror < 0)
  perror (lt_dlerror ());
```

int **lt_dlseterror** (int *errorcode*) Function

When writing your own module loaders, you should use this function to raise errors so that they are propogated through the `lt_dlerror` interface. All of the standard errors used by libltdl are declared in '`ltdl.h`', or you can add more of your own with `lt_dladderror`. This function returns 0 on success.

```
if (lt_dlseterror (LTDL_ERROR_NO_MEMORY) != 0)
  perror (lt_dlerror ());
```

10.6 How to distribute libltdl with your package

Even though libltdl is installed together with libtool, you may wish to include libltdl in the distribution of your package, for the convenience of users of your package that don't have libtool or libltdl installed. In this case, you must decide whether to manually add the `ltdl` objects to your package, or else which flavor of libltdl you want to use: a convenience library or an installable libtool library.

The most simplistic way to add `libltdl` to your package is to copy the source files, '`ltdl.c`' and '`ltdl.h`', to a source directory withing your package and to build and link them along with the rest of your sources. To help you do this, the m4 macros for autoconf are available in '`ltdl.m4`'. You must ensure that they are available in '`aclocal.m4`' before you run autoconf – by appending the contents of '`ltdl.m4`' to '`acinclude.m4`', if you are using automake, or to '`aclocal.m4`' if you are not. Having made the macros available, you must add a call to the '`AC_LIB_LTDL`' macro to your package's '`configure.in`' to perform the configure time checks required to build '`ltdl.o`' correctly. This method has problems if you then try to link the package binaries with an installed libltdl, or a library which depends on libltdl: you may have problems with duplicate symbol definitions.

One advantage of the convenience library is that it is not installed, so the fact that you use libltdl will not be apparent to the user, and it will not overwrite a pre-installed version of libltdl a user might have. On the other hand, if you want to upgrade libltdl for any reason (e.g. a bugfix) you'll have to recompile your package instead of just replacing an installed version of libltdl. However, if your programs or libraries are linked with other libraries that use such a pre-installed version of libltdl, you may get linker errors or run-time crashes. Another problem is that you cannot link the convenience library into more than one libtool library, then link a single program with these libraries, because you may get duplicate symbols. In general you can safely use the convenience library in programs which don't depend on other libraries that might use libltdl too. In order to enable this flavor of libltdl, you should add the line '`AC_LIBLTDL_CONVENIENCE`' to your '`configure.in`', *before* '`AC_PROG_LIBTOOL`'.

In order to select the installable version of libltdl, you should add a call of the macro '`AC_LIBLTDL_INSTALLABLE`' to your '`configure.in`' *before* '`AC_PROG_LIBTOOL`'. This macro will check whether libltdl is already installed and, if not, request the libltdl embedded in your package to be built and installed. Note, however, that no version checking is performed. The user may override the test and determine that the libltdl embedded must be installed, regardless of the existence of another version, using the configure switch '`--enable-ltdl-install`'.

In order to embed libltdl into your package, just add '`--ltdl`' to the `libtoolize` command line. It will copy the libltdl sources to a subdirectory '`libltdl`' in your package. Both macros accept an optional argument to specify the location of the '`libltdl`' directory. By the default both macros assume that it is '`${top_srcdir}/libltdl`'.

Whatever macro you use, it is up to you to ensure that your '`configure.in`' will configure libltdl, using '`AC_CONFIG_SUBDIRS`', and that your '`Makefile`'s will start sub-makes within libltdl's directory, using automake's *SUBDIRS*, for example. Both macros define the shell variables *LIBLTDL*, to the link flag that you should use to link with libltdl, and *INCLTDL*, to the preprocessor flag that you should use to compile with programs that include '`ltdl.h`'. It is up to you to use '`AC_SUBST`' to ensure that this variable will be available in '`Makefile`'s, or add them to variables that are '`AC_SUBST`'ed by default, such as *LIBS* and *CPPFLAGS*.

If you're using the convenience libltdl, *LIBLTDL* will be the pathname for the convenience version of libltdl and *INCLTDL* will be '`-I`' followed by the directory that contains libltdl, both starting with '`${top_builddir}/`' or '`${top_srcdir}/`', respectively.

If you request an installed version of libltdl and one is found[2], *LIBLTDL* will be set to '`-lltdl`' and *INCLTDL* will be empty (which is just a blind assumption that '`ltdl.h`' is somewhere in the include path if libltdl is in the library path). If an installable version of libltdl must be built, its pathname, starting with '`${top_builddir}/`', will be stored in *LIBLTDL*, and *INCLTDL* will be set just like in the case of convenience library.

So, when you want to link a program with libltdl, be it a convenience, installed or installable library, just compile with '`$(INCLTDL)`' and link it with '`$(LIBLTDL)`', using libtool.

---

[2] Even if libltdl is installed, '`AC_LIBLTDL_INSTALLABLE`' may fail to detect it, if libltdl depends on symbols provided by libraries other than the C library. In this case, it will needlessly build and install libltdl.

You should probably also add 'AC_LIBTOOL_DLOPEN' to your 'configure.in' *before* 'AC_PROG_LIBTOOL', otherwise libtool will assume no dlopening mechanism is supported, and revert to dlpreopening, which is probably not what you want.

Avoid using the -static or -all-static switches when linking programs with libltdl. This will not work on all platforms, because the dlopening functions may not be available for static linking.

The following example shows you how to embed the convenience libltdl in your package. In order to use the installable variant just replace 'AC_LIBLTDL_CONVENIENCE' with 'AC_LIBLTDL_INSTALLABLE'. We assume that libltdl was embedded using 'libtoolize --ltdl'.

configure.in:

```
...
dnl Enable building of the convenience library
dnl and set LIBLTDL accordingly
AC_LIBLTDL_CONVENIENCE
dnl Substitute INCLTDL and LIBLTDL in the Makefiles
AC_SUBST(INCLTDL)
AC_SUBST(LIBLTDL)
dnl Check for dlopen support
AC_LIBTOOL_DLOPEN
dnl Configure libtool
AC_PROG_LIBTOOL
dnl Configure libltdl
AC_CONFIG_SUBDIRS(libltdl)
...
```

Makefile.am:

```
...
SUBDIRS = libltdl

INCLUDES = $(INCLTDL)

myprog_LDFLAGS = -export-dynamic
# The quotes around -dlopen below fool automake <= 1.4 into accepting it
myprog_LDADD = $(LIBLTDL) "-dlopen" self "-dlopen" foo1.la
myprog_DEPENDENCIES = $(LIBLTDL) foo1.la
...
```

11 Using libtool with other languages

Libtool was first implemented in order to add support for writing shared libraries in the C language. However, over time, libtool is being integrated with other languages, so that programmers are free to reap the benefits of shared libraries in their favorite programming language.

This chapter describes how libtool interacts with other languages, and what special considerations you need to make if you do not use C.

11.1 Writing libraries for C++

Creating libraries of C++ code should be a fairly straightforward process, because its object files differ from C ones in only three ways:

1. Because of name mangling, C++ libraries are only usable by the C++ compiler that created them. This decision was made by the designers of C++ in order to protect users from conflicting implementations of features such as constructors, exception handling, and RTTI.
2. On some systems, the C++ compiler must take special actions for the dynamic linker to run dynamic (i.e., run-time) initializers. This means that we should not call '`ld`' directly to link such libraries, and we should use the C++ compiler instead.
3. C++ compilers will link some Standard C++ library in by default, but libtool does not know which are these libraries, so it cannot even run the inter-library dependence analyzer to check how to link it in. Therefore, running '`ld`' to link a C++ program or library is deemed to fail. However, running the C++ compiler directly may lead to problems related with inter-library dependencies.

The conclusion is that libtool is not ready for general use for C++ libraries. You should avoid any global or static variable initializations that would cause an "initializer element is not constant" error if you compiled them with a standard C compiler.

There are other ways of working around this problem, but they are beyond the scope of this manual.

Furthermore, you'd better find out, at configure time, what are the C++ Standard libraries that the C++ compiler will link in by default, and explicitly list them in the link command line. Hopefully, in the future, libtool will be able to do this job by itself.

12 Troubleshooting

Libtool is under constant development, changing to remain up-to-date with modern operating systems. If libtool doesn't work the way you think it should on your platform, you should read this chapter to help determine what the problem is, and how to resolve it.

12.1 The libtool test suite

Libtool comes with its own set of programs that test its capabilities, and report obvious bugs in the libtool program. These tests, too, are constantly evolving, based on past problems with libtool, and known deficiencies in other operating systems.

As described in the '`INSTALL`' file, you may run *`make check`* after you have built libtool (possibly before you install it) in order to make sure that it meets basic functional requirements.

12.1.1 Description of test suite

Here is a list of the current programs in the test suite, and what they test for:

'`cdemo-conf.test`'
'`cdemo-exec.test`'
'`cdemo-make.test`'
'`cdemo-static.test`'
'`cdemo-shared.test`'

These programs check to see that the '`cdemo`' subdirectory of the libtool distribution can be configured and built correctly.

The '`cdemo`' subdirectory contains a demonstration of libtool convenience libraries, a mechanism that allows build-time static libraries to be created, in a way that their components can be later linked into programs or other libraries, even shared ones.

The tests '`cdemo-make.test`' and '`cdemo-exec.test`' are executed three times, under three different libtool configurations: '`cdemo-conf.test`' configures '`cdemo/libtool`' to build both static and shared libraries (the default for platforms that support both), '`cdemo-static.test`' builds only static libraries ('`--disable-shared`'), and '`cdemo-shared.test`' builds only shared libraries ('`--disable-static`').

'demo-conf.test'
'demo-exec.test'
'demo-inst.test'
'demo-make.test'
'demo-unst.test'
'demo-static.test'
'demo-shared.test'
'demo-nofast.test'
'demo-pic.test'
'demo-nopic.test'

These programs check to see that the 'demo' subdirectory of the libtool distribution can be configured, built, installed, and uninstalled correctly.

The 'demo' subdirectory contains a demonstration of a trivial package that uses libtool. The tests 'demo-make.test', 'demo-exec.test', 'demo-inst.test' and 'demo-unst.test' are executed four times, under four different libtool configurations: 'demo-conf.test' configures 'demo/libtool' to build both static and shared libraries, 'demo-static.test' builds only static libraries ('--disable-shared'), and 'demo-shared.test' builds only shared libraries ('--disable-static'). 'demo-nofast.test' configures 'demo/libtool' to disable the fast-install mode ('--enable-fast-install=no'). 'demo-pic.test' configures 'demo/libtool' to prefer building PIC code ('--with-pic'), 'demo-nopic.test' to prefer non-PIC code ('--without-pic').

'deplibs.test'

Many systems cannot link static libraries into shared libraries. libtool uses a deplibs_check_method to prevent such cases. This tests checks whether libtool's deplibs_check_method works properly.

'hardcode.test'

On all systems with shared libraries, the location of the library can be encoded in executables that are linked against it see Section 3.3 [Linking executables], page 7. This test checks the conditions under which your system linker hardcodes the library location, and guarantees that they correspond to libtool's own notion of how your linker behaves.

'build-relink.test'

Checks whether variable *shlibpath_overrides_runpath* is properly set. If the test fails and *VERBOSE* is set, it will indicate what the variable should have been set to.

'noinst-link.test'

Checks whether libtool will not try to link with a previously installed version of a library when it should be linking with a just-built one.

'depdemo-conf.test'
'depdemo-exec.test'
'depdemo-inst.test'
'depdemo-make.test'
'depdemo-unst.test'
'depdemo-static.test'
'depdemo-shared.test'
'depdemo-nofast.test'

These programs check to see that the 'depdemo' subdirectory of the libtool distribution can be configured, built, installed, and uninstalled correctly.

The 'depdemo' subdirectory contains a demonstration of inter-library dependencies with libtool. The test programs link some interdependent libraries.

The tests 'depdemo-make.test', 'depdemo-exec.test', 'depdemo-inst.test' and 'depdemo-unst.test' are executed four times, under four different libtool configurations: 'depdemo-conf.test' configures 'depdemo/libtool' to build both static and shared libraries, 'depdemo-static.test' builds only static libraries ('--disable-shared'), and 'depdemo-shared.test' builds only shared libraries ('--disable-static'). 'depdemo-nofast.test' configures 'depdemo/libtool' to disable the fast-install mode ('--enable-fast-install=no'.

'mdemo-conf.test'
'mdemo-exec.test'
'mdemo-inst.test'
'mdemo-make.test'
'mdemo-unst.test'
'mdemo-static.test'
'mdemo-shared.test'

These programs check to see that the 'mdemo' subdirectory of the libtool distribution can be configured, built, installed, and uninstalled correctly.

The 'mdemo' subdirectory contains a demonstration of a package that uses libtool and the system independent dlopen wrapper 'libltdl' to load modules. The library 'libltdl' provides a dlopen wrapper for various platforms (Linux, Solaris, HP/UX etc.) including support for dlpreopened modules (see Section 9.2 [Dlpreopening], page 35).

The tests 'mdemo-make.test', 'mdemo-exec.test', 'mdemo-inst.test' and 'mdemo-unst.test' are executed three times, under three different libtool configurations: 'mdemo-conf.test' configures 'mdemo/libtool' to build both static and shared libraries, 'mdemo-static.test' builds only static libraries ('--disable-shared'), and 'mdemo-shared.test' builds only shared libraries ('--disable-static').

'dryrun.test'

This test checks whether libtool's --dry-run mode works properly.

'assign.test'

Checks whether we don't put break or continue on the same line as an assignment in the libtool script.

'`link.test`'
This test guarantees that linking directly against a non-libtool static library works properly.

'`link-2.test`'
This test makes sure that files ending in '`.lo`' are never linked directly into a program file.

'`nomode.test`'
Check whether we can actually get help for libtool.

'`quote.test`'
This program checks libtool's metacharacter quoting.

'`sh.test`' Checks whether a 'test' command was forgotten in libtool.

'`suffix.test`'
When other programming languages are used with libtool (see Chapter 11 [Other languages], page 52), the source files may end in suffixes other than '`.c`'. This test validates that libtool can handle suffixes for all the file types that it supports, and that it fails when the suffix is invalid.

12.1.2 When tests fail

Each of the above tests are designed to produce no output when they are run via *`make check`*. The exit status of each program tells the '`Makefile`' whether or not the test succeeded.

If a test fails, it means that there is either a programming error in libtool, or in the test program itself.

To investigate a particular test, you may run it directly, as you would a normal program. When the test is invoked in this way, it produces output which may be useful in determining what the problem is.

Another way to have the test programs produce output is to set the *VERBOSE* environment variable to '`yes`' before running them. For example, *`env VERBOSE=yes make check`* runs all the tests, and has each of them display debugging information.

12.2 Reporting bugs

If you think you have discovered a bug in libtool, you should think twice: the libtool maintainer is notorious for passing the buck (or maybe that should be "passing the bug"). Libtool was invented to fix known deficiencies in shared library implementations, so, in a way, most of the bugs in libtool are actually bugs in other operating systems. However, the libtool maintainer would definitely be happy to add support for somebody else's buggy operating system. [I wish there was a good way to do winking smiley-faces in Texinfo.]

Genuine bugs in libtool include problems with shell script portability, documentation errors, and failures in the test suite (see Section 12.1 [Libtool test suite], page 53).

First, check the documentation and help screens to make sure that the behaviour you think is a problem is not already mentioned as a feature.

Then, you should read the Emacs guide to reporting bugs (see section "Reporting Bugs" in *The Emacs Manual*). Some of the details listed there are specific to Emacs, but the principle behind them is a general one.

Finally, send a bug report to the libtool bug reporting address `bug-libtool@gnu.org` with any appropriate *facts*, such as test suite output (see Section 12.1.2 [When tests fail], page 56), all the details needed to reproduce the bug, and a brief description of why you think the behaviour is a bug. Be sure to include the word "libtool" in the subject line, as well as the version number you are using (which can be found by typing *`libtool --version`*).

13 Maintenance notes for libtool

This chapter contains information that the libtool maintainer finds important. It will be of no use to you unless you are considering porting libtool to new systems, or writing your own libtool.

13.1 Porting libtool to new systems

Before you embark on porting libtool to an unsupported system, it is worthwhile to send e-mail to the libtool mailing list `libtool@gnu.org`, to make sure that you are not duplicating existing work.

If you find that any porting documentation is missing, please complain! Complaints with patches and improvements to the documentation, or to libtool itself, are more than welcome.

13.1.1 Information sources

Once it is clear that a new port is necessary, you'll generally need the following information:

canonical system name
: You need the output of `config.guess` for this system, so that you can make changes to the libtool configuration process without affecting other systems.

man pages for `ld` and `cc`
: These generally describe what flags are used to generate PIC, to create shared libraries, and to link against only static libraries. You may need to follow some cross references to find the information that is required.

man pages for `ld.so`, `rtld`, or equivalent
: These are a valuable resource for understanding how shared libraries are loaded on the system.

man page for `ldconfig`, or equivalent
: This page usually describes how to install shared libraries.

output from *`ls -l /lib /usr/lib`*
: This shows the naming convention for shared libraries on the system, including which names should be symbolic links.

any additional documentation
: Some systems have special documentation on how to build and install shared libraries.

If you know how to program the Bourne shell, then you can complete the port yourself; otherwise, you'll have to find somebody with the relevant skills who will do the work. People on the libtool mailing list are usually willing to volunteer to help you with new ports, so you can send the information to them.

To do the port yourself, you'll definitely need to modify the `libtool.m4` macros in order to make platform-specific changes to the configuration process. You should search that file for the `PORTME` keyword, which will give you some hints on what you'll need to change. In general, all that is involved is modifying the appropriate configuration variables (see Section 13.4 [libtool script contents], page 62).

Your best bet is to find an already-supported system that is similar to yours, and make your changes based on that. In some cases, however, your system will differ significantly from every other supported system, and it may be necessary to add new configuration variables, and modify the `ltmain.in` script accordingly. Be sure to write to the mailing list before you make changes to `ltmain.in`, since they may have advice on the most effective way of accomplishing what you want.

13.1.2 Porting inter-library dependencies support

Since version 1.2c, libtool has re-introduced the ability to do inter-library dependency on some platforms, thanks to a patch by Toshio Kuratomi `badger@prtr-13.ucsc.edu`. Here's a shortened version of the message that contained his patch:

The basic architecture is this: in '`libtool.m4`', the person who writes libtool makes sure '`$deplibs`' is included in '`$archive_cmds`' somewhere and also sets the variable '`$deplibs_check_method`', and maybe '`$file_magic_cmd`' when '`deplibs_check_method`' is file_magic.

'`deplibs_check_method`' can be one of five things:

'`file_magic [`*regex*`]`'
: looks in the library link path for libraries that have the right libname. Then it runs '`$file_magic_cmd`' on the library and checks for a match against *regex* using `egrep`. When *file_magic_test_file* is set by '`libtool.m4`', it is used as an argument to '`$file_magic_cmd`' in order to verify whether the regular expression matches its output, and warn the user otherwise.

'`test_compile`'
: just checks whether it is possible to link a program out of a list of libraries, and checks which of those are listed in the output of `ldd`. It is currently unused, and will probably be dropped in the future.

'`pass_all`'
: will pass everything without any checking. This may work on platforms in which code is position-independent by default and inter-library dependencies are properly supported by the dynamic linker, for example, on DEC OSF/1 3 and 4.

'`none`'
: It causes deplibs to be reassigned deplibs=`""`. That way '`archive_cmds`' can contain deplibs on all platforms, but not have deplibs used unless needed.

'`unknown`'
: is the default for all systems unless overridden in '`libtool.m4`'. It is the same as '`none`', but it documents that we really don't know what the correct value should be, and we welcome patches that improve it.

Then in '`ltmain.in`' we have the real workhorse: a little initialization and postprocessing (to setup/release variables for use with eval echo libname_spec etc.) and a case statement that decides which method is being used. This is the real code... I wish I could condense it a little more, but I don't think I can without function calls. I've mostly optimized it (moved things out of loops, etc) but there is probably some fat left. I thought I should stop while I was ahead, work on whatever bugs you discover, etc before thinking about more than obvious optimizations.

13.2 Tested platforms

This table describes when libtool was last known to be tested on platforms where it claims to support shared libraries:

```
%%
%% This is file `.tex',
%% generated with the docstrip utility.
%%
%% The original source files were:
%%
%% fileerr.dtx  (with options: `return')
%%
%% IMPORTANT NOTICE:
%%
%% This file is a generated file from the sources of the `tools' bundle
%% in the LaTeX2e distribution.
%%
%% For the copyright notice see the source file(s).
%%
%% You are not allowed to modify this file.
%%
%% You are allowed to distribute this file if and only if
%% it is distributed with the corresponding source files in
%% the `tools' bundle.
%%
%% For the copying and distribution conditions of the source files,
%% see the file readme.txt distributed with the tools bundle.
%%
 \messageFile ignored
\endinput
%%
%% End of file `.tex'.
```

Note: The vendor-distributed HP-UX `sed`(1) programs are horribly broken, and cannot handle libtool's requirements, so users may report unusual problems. There is no workaround except to install a working `sed` (such as GNU `sed`) on these systems.

Note: The vendor-distributed NCR MP-RAS `cc` programs emits copyright on standard error that confuse tests on size of '`conftest.err`'. The workaround is to specify `CC` when run `configure` with *`CC='cc -Hnocopyr'`*.

13.3 Platform quirks

This section is dedicated to the sanity of the libtool maintainers. It describes the programs that libtool uses, how they vary from system to system, and how to test for them.

Because libtool is a shell script, it can be difficult to understand just by reading it from top to bottom. This section helps show why libtool does things a certain way. Combined with the scripts themselves, you should have a better sense of how to improve libtool, or write your own.

13.3.1 References

The following is a list of valuable documentation references:

- SGI's IRIX Manual Pages, which can be found at `http://techpubs.sgi.com/cgi-bin/infosrch.cgi?cmd=browse&db=man`.
- Sun's free service area (`http://www.sun.com/service/online/free.html`) and documentation server (`http://docs.sun.com/`).
- Compaq's Tru64 UNIX online documentation is at (`http://tru64unix.compaq.com/faqs/publications/pub_page/doc_list.html`) with C++ documentation at (`http://tru64unix.compaq.com/cplus/docs/index.htm`).
- Hewlett-Packard has online documentation at (`http://docs.hp.com/index.html`).
- IBM has online documentation at (`http://www.rs6000.ibm.com/resource/aix_resource/Pubs/`).

13.3.2 Compilers

The only compiler characteristics that affect libtool are the flags needed (if any) to generate PIC objects. In general, if a C compiler supports certain PIC flags, then any derivative compilers support the same flags. Until there are some noteworthy exceptions to this rule, this section will document only C compilers.

The following C compilers have standard command line options, regardless of the platform:

`gcc`

: This is the GNU C compiler, which is also the system compiler for many free operating systems (FreeBSD, GNU/Hurd, GNU/Linux, Lites, NetBSD, and OpenBSD, to name a few).

: The '`-fpic`' or '`-fPIC`' flags can be used to generate position-independent code. '`-fPIC`' is guaranteed to generate working code, but the code is slower on m68k, m88k, and Sparc chips. However, using '`-fpic`' on those chips imposes arbitrary size limits on the shared libraries.

The rest of this subsection lists compilers by the operating system that they are bundled with:

`aix3*`
`aix4*`

: AIX compilers have no PIC flags, since AIX has been ported only to PowerPC and RS/6000 chips.[1]

`hpux10*`

: Use '`+Z`' to generate PIC.

---

[1] All code compiled for the PowerPC and RS/6000 chips (`powerpc-*-*`, `powerpcle-*-*`, and `rs6000-*-*`) is position-independent, regardless of the operating system or compiler suite. So, "regular objects" can be used to build shared libraries on these systems and no special PIC compiler flags are required.

`osf3*` Digital/UNIX 3.x does not have PIC flags, at least not on the PowerPC platform.

`solaris2*`
Use '`-KPIC`' to generate PIC.

`sunos4*` Use '`-PIC`' to generate PIC.

13.3.3 Reloadable objects

On all known systems, a reloadable object can be created by running `ld -r -o` *output*`.o` *input1*`.o` *input2*`.o`. This reloadable object may be treated as exactly equivalent to other objects.

13.3.4 Multiple dependencies

On most modern platforms the order that dependent libraries are listed has no effect on object generation. In theory, there are platforms which require libraries which provide missing symbols to other libraries to listed after those libraries whose symbols they provide.

Particularly, if a pair of static archives each resolve some of the other's symbols, it might be necessary to list one of those archives both before and after the other one. Libtool does not currently cope with this situation well, since dupicate libraries are removed from thr link line.

If you find yourself developing on a host that requires you to list libraries multiple times in order for it to generate correctly linked objects, you can defeat libtool's removal algorithm like this:

```
$ libtool ... -lfoo -lbar -Wl,-lfoo
```

13.3.5 Archivers

On all known systems, building a static library can be accomplished by running `ar cru` *libname*`.a` *obj1*`.o` *obj2*`.o ...`, where the '`.a`' file is the output library, and each '`.o`' file is an object file.

On all known systems, if there is a program named `ranlib`, then it must be used to "bless" the created library before linking against it, with the `ranlib` *libname*`.a` command. Some systems, like Irix, use the `ar ts` command, instead.

13.4 `libtool` script contents

Since version 1.4, the `libtool` script is generated by `configure` (see Section 5.3 [Configuring], page 20). In earlier versions, `configure` achieved this by calling a helper script called '`ltconfig`'. From libtool version 0.7 to 1.0, this script simply set shell variables, then sourced the libtool backend, `ltmain.sh`. `ltconfig` from libtool version 1.1 through 1.3 inlined the contents of `ltmain.sh` into the generated `libtool`, which improved performance on many systems. The tests that '`ltconfig`' used to perform are now kept in '`libtool.m4`' where thay can be written using Autoconf. This has the runtime performance benefits of inlined `ltmain.sh`, *and* improves the build time a little while considerably easing the amount of raw shell code that used to need maintaining.

The convention used for naming variables which hold shell commands for delayed evaluation, is to use the suffix `_cmd` where a single line of valid shell script is needed, and the suffix `_cmds` where multiple lines of shell script **may** be delayed for later evaluation. By convention, `_cmds` variables delimit the evaluation units with the `~` character where necessary.

Here is a listing of each of the configuration variables, and how they are used within `ltmain.sh` (see Section 5.3 [Configuring], page 20):

AR Variable

The name of the system library archiver.

CC Variable

The name of the C compiler used to configure libtool.

LD Variable

The name of the linker that libtool should use internally for reloadable linking and possibly shared libraries.

NM Variable

The name of a BSD-compatible `nm` program, which produces listings of global symbols in one the following formats:

address `C` *global-variable-name*
address `D` *global-variable-name*
address `T` *global-function-name*

RANLIB Variable

Set to the name of the ranlib program, if any.

allow_undefined_flag Variable

The flag that is used by '`archive_cmds`' in order to declare that there will be unresolved symbols in the resulting shared library. Empty, if no such flag is required. Set to '`unsupported`' if there is no way to generate a shared library with references to symbols that aren't defined in that library.

always_export_symbols Variable

Whether libtool should automatically generate a list of exported symbols using *export_symbols_cmds* before linking an archive. Set to '`yes`' or '`no`'. Default is '`no`'.

archive_cmds Variable
archive_expsym_cmds Variable
old_archive_cmds Variable

Commands used to create shared libraries, shared libraries with '`-export-symbols`' and static libraries, respectively.

old_archive_from_new_cmds Variable

If the shared library depends on a static library, '`old_archive_from_new_cmds`' contains the commands used to create that static library. If this variable is not empty, '`old_archive_cmds`' is not used.

old_archive_from_expsyms_cmds Variable

If a static library must be created from the export symbol list in order to correctly link with a shared library, '`old_archive_from_expsyms_cmds`' contains the commands needed to create that static library. When these commands are executed, the variable *soname* contains the name of the shared library in question, and the *$objdir/$newlib* contains the path of the static library these commands should build. After executing these commands, libtool will proceed to link against *$objdir/$newlib* instead of *soname*.

build_libtool_libs Variable

Whether libtool should build shared libraries on this system. Set to '`yes`' or '`no`'.

build_old_libs Variable

Whether libtool should build static libraries on this system. Set to '`yes`' or '`no`'.

compiler_c_o Variable

Whether the compiler supports the `-c` and `-o` options simultaneously. Set to '`yes`' or '`no`'.

compiler_o_lo Variable

Whether the compiler supports compiling directly to a ".lo" file, i.e whether object files do not have to have the suffix ".o". Set to '`yes`' or '`no`'.

dlopen_support Variable

Whether `dlopen` is supported on the platform. Set to '`yes`' or '`no`'.

dlopen_self Variable

Whether it is possible to `dlopen` the executable itself. Set to '`yes`' or '`no`'.

dlopen_self_static Variable

Whether it is possible to `dlopen` the executable itself, when it is linked statically ('`-all-static`'). Set to '`yes`' or '`no`'.

echo Variable

An `echo` program which does not interpret backslashes as an escape character.

exclude_expsyms Variable

List of symbols that should not be listed in the preloaded symbols.

export_dynamic_flag_spec Variable

Compiler link flag that allows a dlopened shared library to reference symbols that are defined in the program.

export_symbols_cmds Variable

Commands to extract exported symbols from *libobjs* to the file *export_symbols*.

extract_expsyms_cmds Variable

Commands to extract the exported symbols list from a shared library. These commands are executed if there is no file *$objdir/$soname-def*, and should write the names of the exported symbols to that file, for the use of '`old_archive_from_expsyms_cmds`'.

fast_install Variable

Determines whether libtool will privilege the installer or the developer. The assumption is that installers will seldom run programs in the build tree, and the developer will seldom install. This is only meaningful on platforms in which *shlibpath_overrides_runpath* is not '`yes`', so *fast_install* will be set to '`needless`' in this case. If *fast_install* set to '`yes`', libtool will create programs that search for installed libraries, and, if a program is run in the build tree, a new copy will be linked on-demand to use the yet-to-be-installed libraries. If set to '`no`', libtool will create programs that use the yet-to-be-installed libraries, and will link a new copy of the program at install time. The default value is '`yes`' or '`needless`', depending on platform and configuration flags, and it can be turned from '`yes`' to '`no`' with the configure flag '`--disable-fast-install`'.

finish_cmds Variable

Commands to tell the dynamic linker how to find shared libraries in a specific directory.

finish_eval Variable

Same as *finish_cmds*, except the commands are not displayed.

fix_srcfile_path Variable

Expression to fix the shell variable $srcfile for the compiler.

global_symbol_pipe Variable

A pipeline that takes the output of *NM*, and produces a listing of raw symbols followed by their C names. For example:

```
$ eval "$NM progname | $global_symbol_pipe"
D symbol1 C-symbol1
T symbol2 C-symbol2
C symbol3 C-symbol3
...
$
```

The first column contains the symbol type (used to tell data from code on some platforms), but its meaning is system dependent.

global_symbol_to_cdecl Variable

A pipeline that translates the output of *global_symbol_pipe* into proper C declarations. On platforms whose linkers differentiate code from data, such as HP/UX, data symbols will be declared as such, and code symbols will be declared as functions. On platforms that don't care, everything is assumed to be data.

hardcode_action Variable

Either '`immediate`' or '`relink`', depending on whether shared library paths can be hardcoded into executables before they are installed, or if they need to be relinked.

hardcode_direct Variable

Set to '`yes`' or '`no`', depending on whether the linker hardcodes directories if a library is directly specified on the command line (such as '*dir*`/lib`*name*`.a`') when *hardcode_libdir_flag_spec* is specified.

hardcode_into_libs Variable

Whether the platform supports hardcoding of run-paths into libraries. If enabled, linking of programs will be much simpler but libraries will need to be relinked during installation. Set to '`yes`' or '`no`'.

hardcode_libdir_flag_spec Variable

Flag to hardcode a *libdir* variable into a binary, so that the dynamic linker searches *libdir* for shared libraries at runtime. If it is empty, libtool will try to use some other hardcoding mechanism.

hardcode_libdir_separator Variable

If the compiler only accepts a single *hardcode_libdir_flag*, then this variable contains the string that should separate multiple arguments to that flag.

hardcode_minus_L Variable

Set to '`yes`' or '`no`', depending on whether the linker hardcodes directories specified by '`-L`' flags into the resulting executable when *hardcode_libdir_flag_spec* is specified.

hardcode_shlibpath_var Variable

Set to '`yes`' or '`no`', depending on whether the linker hardcodes directories by writing the contents of '`$shlibpath_var`' into the resulting executable when *hardcode_libdir_flag_spec* is specified. Set to '`unsupported`' if directories specified by '`$shlibpath_var`' are searched at run time, but not at link time.

host Variable
host_alias Variable

For information purposes, set to the specified and canonical names of the system that libtool was configured for.

include_expsyms Variable

List of symbols that must always be exported when using *export_symbols*.

libext Variable

The standard old archive suffix (normally "a").

libname_spec Variable

The format of a library name prefix. On all Unix systems, static libraries are called '`lib`*name*`.a`', but on some systems (such as OS/2 or MS-DOS), the library is just called '*name*`.a`'.

library_names_spec Variable

A list of shared library names. The first is the name of the file, the rest are symbolic links to the file. The name in the list is the file name that the linker finds when given '`-l`*name*'.

link_all_deplibs Variable

Whether libtool must link a program against all its dependency libraries. Set to '`yes`' or '`no`'. Default is '`unknown`', which is a synonym for '`yes`'.

link_static_flag Variable

Linker flag (passed through the C compiler) used to prevent dynamic linking.

need_lib_prefix Variable

Whether libtool should automatically prefix module names with 'lib'. Set to '`yes`' or '`no`'. By default, it is '`unknown`', which means the same as '`yes`', but documents that we are not really sure about it. '`yes`' means that it is possible both to `dlopen` and to link against a library without 'lib' prefix, i.e. it requires *hardcode_direct* to be '`yes`'.

need_version Variable

Whether versioning is required for libraries, i.e. whether the dynamic linker requires a version suffix for all libraries. Set to '`yes`' or '`no`'. By default, it is '`unknown`', which means the same as '`yes`', but documents that we are not really sure about it.

need_locks Variable

Whether files must be locked to prevent conflicts when compiling simultaneously. Set to '`yes`' or '`no`'.

no_builtin_flag Variable

Compiler flag to disable builtin functions that conflict with declaring external global symbols as `char`.

no_undefined_flag Variable

The flag that is used by '`archive_cmds`' in order to declare that there will be no unresolved symbols in the resulting shared library. Empty, if no such flag is required.

objdir Variable

The name of the directory that contains temporary libtool files.

objext Variable
: The standard object file suffix (normally "o").

pic_flag Variable
: Any additional compiler flags for building library object files.

postinstall_cmds Variable
old_postinstall_cmds Variable
: Commands run after installing a shared or static library, respectively.

postuninstall_cmds Variable
old_postuninstall_cmds Variable
: Commands run after uninstalling a shared or static library, respectively.

reload_cmds Variable
reload_flag Variable
: Commands to create a reloadable object.

runpath_var Variable
: The environment variable that tells the linker which directories to hardcode in the resulting executable.

shlibpath_overrides_runpath Variable
: Indicates whether it is possible to override the hard-coded library search path of a program with an environment variable. If this is set to no, libtool may have to create two copies of a program in the build tree, one to be installed and one to be run in the build tree only. When each of these copies is created depends on the value of `fast_install`. The default value is '`unknown`', which is equivalent to '`no`'.

shlibpath_var Variable
: The environment variable that tells the dynamic linker where to find shared libraries.

soname_spec Variable
: The name coded into shared libraries, if different from the real name of the file.

striplib Variable
old_striplib Variable
: Command to strip a shared (`striplib`) or static (`old_striplib`) library, respectively. If these variables are empty, the strip flag in the install mode will be ignored for libraries (see Section 4.4 [Install mode], page 18).

sys_lib_dlsearch_path_spec Variable
: Expression to get the run-time system library search path. Directories that appear in this list are never hard-coded into executables.

sys_lib_search_path_spec Variable

Expression to get the compile-time system library search path. This variable is used by libtool when it has to test whether a certain library is shared or static. The directories listed in *shlibpath_var* are automatically appended to this list, every time libtool runs (i.e., not at configuration time), because some linkers use this variable to extend the library search path. Linker switches such as `-L` also augment the search path.

thread_safe_flag_spec Variable

Linker flag (passed through the C compiler) used to generate thread-safe libraries.

version_type Variable

The library version numbering type. One of '`libtool`', '`freebsd-aout`', '`freebsd-elf`', '`irix`', '`linux`', '`osf`', '`sunos`', '`windows`', or '`none`'.

whole_archive_flag_spec Variable

Compiler flag to generate shared objects from convenience archives.

wl Variable

The C compiler flag that allows libtool to pass a flag directly to the linker. Used as: `${wl}`*some-flag*.

Variables ending in '`_cmds`' or '`_eval`' contain a '`~`'-separated list of commands that are `eval`ed one after another. If any of the commands return a nonzero exit status, libtool generally exits with an error message.

Variables ending in '`_spec`' are `eval`ed before being used by libtool.

13.5 Cheap tricks

Here are a few tricks that you can use in order to make maintainership easier:

- When people report bugs, ask them to use the '`--config`', '`--debug`', or '`--features`' flags, if you think they will help you. These flags are there to help you get information directly, rather than having to trust second-hand observation.
- Rather than reconfiguring libtool every time I make a change to `ltmain.in`, I keep a permanent `libtool` script in my *PATH*, which sources `ltmain.in` directly.

  The following steps describe how to create such a script, where `/home/src/libtool` is the directory containing the libtool source tree, `/home/src/libtool/libtool` is a libtool script that has been configured for your platform, and `~/bin` is a directory in your *PATH*:

  ```
  trick$ cd ~/bin
  trick$ sed '/^# ltmain\.sh/q' /home/src/libtool/libtool > libtool
  trick$ echo '. /home/src/libtool/ltmain.in' >> libtool
  trick$ chmod +x libtool
  trick$ libtool --version
  ltmain.sh (GNU @PACKAGE@) @VERSION@@TIMESTAMP@
  trick$
  ```

The output of the final 'libtool --version' command shows that the ltmain.in script is being used directly. Now, modify ~/bin/libtool or /home/src/libtool/ltmain.in directly in order to test new changes without having to rerun configure.

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent" is called "Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.1
> or any later version published by the Free Software Foundation;
> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

E

F

G

H

I

L

R

S

T

U

V

W

Table of Contents