Build a shared library using GCC

Content

Tested on

Debian (Etch, Lenny, Squeeze)

Ubuntu (Hardy, Intrepid, Jaunty, Karmic, Lucid, Maverick, Natty, Oneiric, Precise, Quantal)

Objective

To build a shared library using GCC

Background

Programs can be linked against libraries either at compile time or at run time.
An advantage of linking at run time is that a single copy of the library can be shared between many programs, both on disc and in memory. Libraries suitable for use in this way are known as shared libraries.

On modern Linux-based systems, shared libraries differ from static ones in the following respects:

- they are ELF files (as opposed to archives compatible with the `ar` program),
- they have a dynamic symbol table (in addition to a static table), and
- the code within them must be position-independent.

For these reasons, some adjustments to the build process are needed to create a shared library instead of a static one.

Scenario

Suppose that you are building a library named `libqux` which is written in C. There are three source files: `foo.c`, `bar.c` and `baz.c`.

The current version number of `libqux` is 1.5.0. It is fully backward-compatible with the previous version, 1.4.1, which had an soname of `libqux.so.1`.

Method

Overview

The method described here has three steps:

1. Choose an soname (if required).
2. Compile the source code using the `-fPIC` option.
3. Link the object code using the `-fPIC` and `-shared` options.

Choose an soname (if required)

An soname (‘shared object name’) is a label used when declaring shared library dependencies. Each executable contains a list of shared libraries that it needs in order to execute. Shared libraries can similarly declare dependencies on other shared libraries. This can be done using pathnames, but if the required library has an soname then that will be used in preference.

Typically the pathname of a library will change whenever a new version is installed, whereas the soname should change only when the new version is incompatible with its predecessors to the extent that it cannot be used their place. It follows that when dependencies are declared using sonames, the library used at runtime need not be an exact match for the one present at build time:

- For a given library with a given soname, only the most recent version need be installed.
- Where there is a need for two versions of the same library to be installed alongside each other, they can be distinguished because they have different sonames.

It is the degree of binary compatibility which determines whether the soname should change. For example, new functions can be added without breaking backward compatibility, but you cannot normally change the prototype of an existing function, nor do anything that could change the layout of a data structure. You should also consider changes made to the high-level behaviour of the library, as these can have an equally significant effect on backwards compatibility.

In this particular instance, version 1.4.1 of `libqux` had an soname of `libqux.so.1`. Since version 1.5.0 is backwards-compatibile it can use the same soname. If this had not been the case then the soname would have needed to change, most likely to `libqux.so.2`.

Compile the source code using the -fPIC option

Object code intended for use in a shared library must be ‘position-independent’, meaning that it can execute without first being modified to account for where it has been loaded in memory. It remains necessary to allow for the location of *other* libraries, but any internal references are required to be position-independent.

GCC can be instructed to generate position independent code using the `-fPIC` option:

```
gcc -c -fPIC -o foo.o foo.c
gcc -c -fPIC -o bar.o bar.c
gcc -c -fPIC -o baz.o baz.c
```

This option is not enabled by default because it tends to cause some loss of performance, and for purposes other than building shared libraries it is often not necessary.

Link the object code using the -fPIC and -shared options

The default behaviour of the `gcc` and `g++` commands when linking is to produce an executable program. They can be instructed to produce a shared library instead by means of `-shared` option:

```
gcc -shared -fPIC -Wl,-soname,libqux.so.1 -o libqux.so.1.5.0 foo.o bar.o baz.o -lc
```

The `-fPIC` option is needed when linking as it was when compiling to ensure that any code added by the linker is compatible with code previously generated by the compiler.

The `-Wl` option passes a comma-separated list of arguments to the linker. As its name suggests, `-soname` specifies the required soname. If these options are omitted then the library will not have an soname.

The `ldconfig` manpage recommends explicitly linking against `libc`, which has been done above using the `-l` option (`-lc`).

Testing

One way to test the library is to install it in a directory on the library search path. `/usr/local/lib` is usually the most appropriate choice. You will need to create softlinks corresponding to the soname of the library, and the name used to refer to the library when building the executable, if these are different from the filename:

```
ln -s libqux.so.1.5.0 libqux.so.1
ln -s libqux.so.1.5.0 libqux.so
```

A partial alternative is to run `ldconfig`, which automatically creates the first of the above softlinks but not the second. However you do it, this method of testing normally requires administrative privileges. Once installed, it should be possible to link against the library using `-l`:

```
gcc main.c -lqux
```

If you cannot or do not want to move the library to `/usr/local/lib` then it is possible to link against the library *in situ*. At build time this can be done by listing the pathname of the library as an argument to gcc without use of the `-l` option:

```
gcc main.c libqux.so.1.5.0
```

At load time you will need to add the relevant directory to the library search path. This can be done by setting the environment variable `LD_LIBRARY_PATH`, for example:

```
export LD_LIBRARY_PATH=`pwd`
```

As above, you will need to create a softlink corresponding to the soname of the library. If there is a need to search multiple directories then they should be specified as a colon-separated list in `LD_LIBRARY_PATH`.

Alternatives

Using GNU Libtool

Libtool is part of GNU Autotools. Its purpose is to simplify the process of building shared libraries, particularly those intended for use on multiple platforms. For example, for the scenario described above you could use the following sequence of commands:

```
libtool --mode=compile gcc -c foo.c
libtool --mode=compile gcc -c bar.c
libtool --mode=compile gcc -c baz.c
libtool --mode=link gcc -o libqux.la foo.lo bar.lo baz.lo -rpath /usr/local/lib -version-info 6:0:5
```

You may not need to these commands explicitly, because Libtool is often used in conjunction with Automake which has the ability to generate them automatically, but it is equally suitable for use as a stand-alone utility if that suits your purpose.

Be aware that Libtool requires the use of a specific numbering scheme for specifying the interface version (passed using the `-version-info` option above), and that this should almost certainly not be equal to the release version. The Libtool manual describes when and how these values should be changed.

Further reading

- Program Library HOWTO, David A Wheeler
- Libtool's versioning system, *GNU Libtool Manual*, GNU Project
- Vaughan *et al*, Library Versioning, *GNU Autoconf, Automake and Libtool*
- ldconfig(8) (Ubuntu manpage)

Tags: gcc | make