Autotools Training

Practical Labs

bootlin

https://bootlin.com

July 11, 2025

About this document

Updates to this document can be found on https://bootlin.com/doc/training/autotools.

This document was generated from LaTeX sources found on https://github.com/bootlin/training-materials.

More details about our training sessions can be found on https://bootlin.com/training.

Copying this document

Corrections, suggestions, contributions and translations are welcome!

Training setup

Download files and directories used in practical labs

Install lab data

For the different labs in this course, your instructor has prepared a set of data (kernel images, kernel configurations, root filesystems and more). Download and extract its tarball from a terminal:

```
$ cd
$ wget https://bootlin.com/doc/training/autotools/autotools-labs.tar.xz
$ tar xvf autotools-labs.tar.xz
```

Lab data are now available in an `autotools-labs` directory in your home directory. This directory contains directories and files used in the various practical labs. It will also be used as working space, in particular to keep generated files separate when needed.

Update your distribution

To avoid any issue installing packages during the practical labs, you should apply the latest updates to the packages in your distro:

```
$ sudo apt update
$ sudo apt dist-upgrade
```

You are now ready to start the real practical labs!

Install extra packages

Feel free to install other packages you may need for your development environment. In particular, we recommend to install your favorite text editor and configure it to your taste. The favorite text editors of embedded Linux developers are of course *Vim* and *Emacs*, but there are also plenty of other possibilities, such as Visual Studio Code[1], *GEdit*, *Qt Creator*, *CodeBlocks*, *Geany*, etc.

It is worth mentioning that by default, Ubuntu comes with a very limited version of the `vi` editor. So if you would like to use `vi`, we recommend to use the more featureful version by installing the `vim` package.

More guidelines

Can be useful throughout any of the labs

- Read instructions and tips carefully. Lots of people make mistakes or waste time because they missed an explanation or a guideline.
- Always read error messages carefully, in particular the first one which is issued. Some people stumble on very simple errors just because they specified a wrong file path and didn't pay enough attention to the corresponding error message.
- Never stay stuck with a strange problem more than 5 minutes. Show your problem to your colleagues or to the instructor.
- You should only use the `root` user for operations that require super-user privileges, such as: mounting a file system, loading a kernel module, changing file ownership, configuring the network. Most regular tasks (such as downloading, extracting sources, compiling...) can be done as a regular user.

[1] This tool from Microsoft is Open Source! To try it on Ubuntu: `sudo snap install code --classic`

- If you ran commands from a root shell by mistake, your regular user may no longer be able to handle the corresponding generated files. In this case, use the `chown -R` command to give the new files back to your regular user.
  Example: `$ sudo chown -R myuser.myuser linux/`

Usage of existing *autotools* projects

Objectives:

- *First build of an* autotools *package*
- *Out of tree build and cross-compilation*
- *Overriding cache variables*
- *Using* autoreconf

Download ethtool

Go to the `\$HOME/autotools-labs/` directory.

To start our exploration of *autotools*, we'll use the **ethtool** set of programs. Download the 5.17 version from the project home page at https://www.kernel.org/pub/software/network/ethtool/ and extract it.

Exploration of the sources

Inside the sources, look at the various files available, and pay special attention to `configure.ac` and `Makefile.am`. Even though we haven't yet discussed their syntax and contents, try to get some idea about what they could be doing.

Compare their length with the length of their corresponding generated files `configure` and `Makefile.in`. Also, check if there is already a `Makefile` or not in the project.

Read the output of `./configure --help`.

First build

Run `./configure`, and look at the output. You may need to install the `libmnl-dev` package if it is not already installed on your system. Also look at the contents of the `config.log` file.

Then, run the build with `make`.

Finally, try to run the installation with `make install`. As you can see, it fails with *Permission denied* messages, because it tries to install to `/usr/local`, which is the default *prefix*.

Since we don't want to install to `/usr/local`, let's configure *ethtool* to install to a different place:

```
mkdir $HOME/sys
./configure --prefix=$HOME/sys/
make
make install
```

This time, in `$HOME/sys`, you should have the *ethtool* program an man pages installed.

Out of tree build and cross-compiling

Now, create a separate build directory, say `$HOME/ethtool-arm/`, and move to this directory. Then, call the `configure` script of *ethtool*:

```
../ethtool-5.17/configure
```

This will abort with an error: it doesn't want to do *out of tree* build if the source tree has been configured (which we did in the previous section of this lab). So as suggested, clean up the *ethtool* source tree by running `make distclean`.

To cross-compile *ethtool*, we'll first have to install a cross-compiler:

```
apt install gcc-arm-linux-gnueabihf
```

Then you can do:

```
../ethtool-5.17/configure --host=arm-linux-gnueabihf
```

Start the build, and verify that the cross-compiler is used. The build will fail because *libmnl* isn't available on our system, cross-compiled for ARM. Let's build without *netlink* support to avoid the *libmnl* dependency.

```
../ethtool-5.17/configure --host=arm-linux-gnueabihf --disable-netlink
```

Overriding cache variables

If you look at the *configure* output, you can see:

```
checking for strtol... yes
```

The *configure* script is checking for the availability of the `strtol()` function. Now let's pretend for some reason that the detected value is not appropriate (it's obviously not the case for such a simple test, but we need to take an example!). We want to override this test by passing the appropriate value as an environment variable to the `configure` script.

Before doing this change, look at the `config.h` *configuration header* and see the value of the definition related to `strtol`. Note that *ethtool* decided to use a custom name for the *configuration header* file, and it's actually named `ethtool-config.h`.

Read the `config.log`, and identify which variable can be used to override the result of the `strtol` test. Pass it in the environment of `./configure`, and check that *configure* outputs:

```
checking for strtol... (cached) no
```

You can also check in `ethtool-config.h` the new value for definition related to `strtol`.

However, it's interesting to notice that in practice the *ethtool* source code assumes `strtol` is present, without taking into account the `HAVE_STRTOL` definition. So if the function was really absent, there would be a build failure.

Autoreconfiguring

Now, let's look at another piece of software, which isn't distributed with pre-generated `configure` and `Makefile.in` files.

Go back to your `$HOME` directory (or another directory you created for this training session), and use *Git* to fetch the source code of a library called *libconfuse*:

```
git clone https://github.com/martinh/libconfuse
```

Note: you may need to install *Git*, using `apt install git`.

Look at the *libconfuse* source code, and see that you only have `configure.ac` but not the generated `configure` script, and only the `Makefile.am` files, and not the generated `Makefile.in` files.

Now, run `autoreconf -i`, and look at which files where generated.

If you get issues when running `autoreconf`, it is most likely because your system doesn't have the *autotools* installed. In this case, run:

```
apt install autoconf automake libtool autopoint gettext
```


(Note: `autopoint` and `gettext` are needed because this package uses internationalization features provided by *gettext*, which we won't cover in this training.)

You can now build and install *libconfuse* in `$HOME/sys`. For good measure, don't forget to disable building the examples: look at `./configure --help` to know how to do that.

When building, you should see a message `flex: command not found`. Indeed, this tool is not present on our system. Interestingly, if you look back at the output of the `./configure` script, it did check for `flex`, concluded it wasn't available, but did not error out. As you can see not all `configure` scripts are properly written!

Install *flex*:

```
apt install flex
```

Re-run `./configure`, and restart the build.

Autotools basics

Objectives:

- *Your first* `configure.ac`
- *Adding and building a program*
- *Going further:* `autoscan` *and* `make dist`

Create your first `configure.ac`

In `$HOME` (or the directory you created for this training), create a new folder `project` for your first *autotools* project.

In order to keep a good understanding of real source files as opposed to generated files, we'll put our project under version control, using *Git*. Run `git init` to initialize a new repository.

Create the absolutely minimal `configure.ac` using just `AC_INIT` and `AC_OUTPUT`. Use `git add configure.ac` to add this file to the Git repository, and `git commit -s -m "Initial configure.ac"` to commit this first `configure.ac`.

Use `autoreconf -i`, look at which files where generated. Thanks to `git status`, you can see which of those files are not under version control.

Test your shiny new `configure` script. How much does it do? Look at the new files that have been generated.

Now, prepare your project to support C source code, by adding `AC_PROG_CC` to your `configure.ac`. Run `autoreconf -i` again, and restart `./configure`. It should do a lot more tests now!

Adding and building a program

Create a very simple hello world C program in a file called `hello.c`:

```
#include <stdio.h>

int main(void) {
   printf("Hello World\n");
   return 0;
}
```

In order to build this file, you'll have to adjust your `configure.ac` to initialize `automake` and create a `Makefile.am` file with the necessary code to build one program.

Run `autoreconf -i`, and once it *autoreconfs* properly, build your project!

You can also install it, after re-configuring it with `--prefix=$HOME/sys`.

As you can see, doing your first *autotools* project is not very complicated.

Going further

Run `autoscan`, read the generated `configure.scan` and compare it to your `configure.ac`. What are the differences?

Run `make dist`, and look at the tarball that is generated. Your project is ready to be released!

Now, run `git add` on `configure.ac`, `Makefile.am` and `hello.c`, and do a new commit: `git commit -s -m "Add a real program"`. If you run `git status`, you can see that there are really a lot of files generated by *autotols*, so add a `.gitignore` file to tell Git to simply ignore them. This file should contain something like:

```
.deps/
Makefile
Makefile.in
aclocal.m4
autom4te.cache/
compile
config.log
config.status
configure
depcomp
hello
hello.o
install-sh
missing
```

The files `configure.scan` and `autoscan.log` can simply be removed, they would only get re-created if we run `autoscan` again.

Commit your `.gitignore` file

```
git add .gitignore
git commit -s -m ``Add gitignore file''
```

Now at any time you can do `git clean -xdf` to ask Git to remove all the files that are not under version control. This allows to easily get rid of all the files generated by the *autotools* and see more clearly what's part of your project.

Autotools advanced

Objectives:

- *Use* `AC_ARG_ENABLE` *and* `config.h`
- *Implement a shared library*
- *Switch to multiple directories*
- *Make the compilation of programs conditional*
- *Use* `pkg-config`

Use AC_ARG_ENABLE and config.h

Continue to work on the project started in the previous lab. Start by adding a `--enable-message` option in the `configure.ac` by using `AC_ARG_ENABLE`. The purpose of this option, which should be enabled by default, is to allow you to enable or disable the printing of the *Hello World* message.

You will have to combine a *configuration header*, and `AC_DEFINE` to let the C source code know whether the option was enabled or not. `AC_DEFINE` could for example define a macro named `WANTS_HELLO_WORLD`.

Once you are done, test that running `./configure` or `./configure --enable-message` gives a `config.h` file with `WANTS_HELLO_WORLD` defined, and that `./configure --disable-message` gives a `config.h` without `WANTS_HELLO_WORLD`.

Change the `hello.c` program to use `WANTS_HELLO_WORLD`. Build your `hello` program with `--enable-message` and `--disable-message` consecutively, and check that in the first case, `Hello World` is display, and that in the second case `Hello World` is not displayed:

```
$ ./configure --enable-message
$ make
$ ./hello
Hello World
$ ./configure --disable-message
$ make
$ ./hello
$
```

Implement a shared library

For the purpose of this training, we'll implement a small library called `libhello`, which provides a single function `show_msg()` responsible for displaying the `Hello World` message. Our program will link against this library and use the function it provides.

So create a file named `core.c`, with this function `show_msg()` (it should continue to use the `WANTS_HELLO_WORLD` macro defined in the previous section).

Create a header file named `hello.h`, which contains the prototype of the `show_msg()` function.

Then, change the `hello.c` program so that it calls the `show_msg()` function.

Once the source preparation is done, it's time to adapt the build system:

1. Adapt the `configure.ac` script to initialize `libtool`
2. Change the `Makefile.am` to:
   - build the `libhello` library from the `core.c` file
   - declare the library version
   - install the `hello.h` header
   - make the `hello` program link against the `libhello` library

Once this is done, *autoreconf* your project, run `configure`, build, and run the `hello` program.

Configure it with `--prefix=$HOME/sys`, and install it there. You should see the library being installed in `$HOME/sys/lib` and the header file in `$HOME/sys/include`. By running:

```
readelf -d $HOME/sys/bin/hello
```

You can verify that `hello` is indeed linked against `libhello`.

To finish up this part, if you look back at the `autoreconf -i` output, it complained about `configure.ac` not using `AC_CONFIG_MACRO_DIR`. So as explained in the slides, make the needed changes to `configure.ac` and `Makefile.am`.

Switch to multiple directories

Even though our project is small, it's time to experiment with subdirectories. Since we're modern, we'll use *non-recursive make*.

Move `core.c` and `hello.h` to a folder called `lib/`, and move `hello.c` to a folder called `src/`.

Adjust the main `Makefile.am` accordingly, and make sure everything continues to build correctly.

Hint: you will have to add a `hello_CPPFLAGS` variable.

Make the compilation of programs conditional

Since some people may not be interested in building the `hello` program, we'll make this optional. Create a new `--enable-programs` option in `configure.ac`, which should be enabled by default.

Then, use an *automake* conditional to build the `hello` program only if enabled. Verify that when you pass `--disable-programs`, the `hello` program is not built.

Use `pkg-config`

Now we will make our program rely on the `libconfig` library. You can install this library on your system, and its development files by running:

```
apt install libconfig-dev
```

Now, change `src/hello.c` to do the following:

```
#include <libconfig.h>
#include "hello.h"

int main(void)
{
   config_t cfg;
   config_init(&cfg);
   show_msg();
   return 0;
}
```


It doesn't do anything useful, but calls one function of the `libconfig` library, `config_init`, which is enough for our demonstration.

If you try to build the program, it will fail with the following error:

```
hello.c:7: undefined reference to `config_init'
```

This is because we are not yet linking with the `libconfig` library. We will use `pkg-config` to detect it and use the appropriate linker flags. To do that, use `PKG_CHECK_MODULES` in `configure.ac` when programs are enabled, and adjust your `Makefile.am` to use the compiler and linker flags provided by the `PKG_CHECK_MODULES` macro.

If everything works fine, you should see the `./configure` script detecting `pkg-config`:

```
$ ./configure
[...]
checking pkg-config is at least version 0.9.0... yes
checking for LIBCONFIG... yes
[...]
```

And finally the `hello` binary being linked with `libconfig`:

```
$ make
[...]
libtool: link: gcc -g -O2 -o .libs/hello src/hello-hello.o  ./.libs/libhello.so -lconfig
[...]
```

Congratulations, your program is now properly using the `libconfig` library!

Going further

Implement the use of *silent rules* as explained in the slides, and experiment with `make V=1` vs. `make V=0`.

Add a README file to your project, and make sure it is properly distributed as part of the tarball generated by `make dist`

Use `AC_CONFIG_AUX_DIR` to store the *auxiliary files* in a custom directory.

