Using Automake in the Groff project

by

Bertrand Garrigues

©2014, 2017 Free Software Foundation

Contents

1. Overview, the initial build 1
   - *1.1. First build* 1
   - *1.2. Automake in the autotools process* 2
   - *1.3. Modification of autotools files* 2
2. Building a program 3
   - *2.1. A program and its source files* 3
   - *2.2. Linking against a library* 4
   - *2.3. Preprocessor flags* 4
   - *2.4. Cleaning* 4
   - *2.5. Dependencies* 4
   - *2.6. Scripts* 5
3. Non-recursive make schema 5
   - *3.1. 1st possibility: make recursion* 6
   - *3.2. Non-recursive make used by the Groff project* 6
4. Installing data 7
   - *4.1. A simple case* 7
   - *4.2. Dealing with generated files* 8
5. Extending Automake's rules 8
   - *5.1. Local clean rules* 8
   - *5.2. Local install/uninstall rules and hooks* 9

Using Automake in the Groff project

by
Bertrand Garrigues

This is a quick overview of how to use ‘automake’ in the groff project, and is intended to help the developers and contributors find their way when they have to make changes to the sources files or to the data that are installed. If you need more details on ‘automake’, here are some reading suggestions:

- The Automake Manual:
  https://www.gnu.org/software/automake/manual/automake.html
- A book by John Calcote, with good practical examples:
  http://fsmsh.com/2753
- This site, by Diego Petteno, with good practical examples too:
  https://autotools.io/index.html

1. Overview, the initial build

1.1. First build

Groff integrates the ‘gnulib’ and uses its ‘bootstrap’ script. When compiling from the git repository, you should first invoke this script:

```
$ ./bootstrap
```

This will:

- Clone the gnulib repository as a git submodule in ‘gnulib’, add the needed gnulib sources files in ‘lib’, add the needed gnulib m4 macros in ‘gnulib_m4’.
- Invoke autoreconf that will call all the ‘GNU autotools’ (‘aclocal’, ‘autoheader’, ‘autoconf’, ‘automake’) in the right order for creating the following files:
  – INSTALL (a symlink to gnulib’s INSTALL file)
  – Makefile.in
  – aclocal.m4
  – autom4te.cache/
  – build-aux/ (that contains all the helper scripts)
  – configure
  – src/include/config.hin

The file aclocal.m4 is generated and the groff m4 macros are included via the acinclude.m4 file.

At this point you can invoke the ‘configure’ script and call ‘make’ to build the groff

project. You can do it in the source tree:

```
$ ./configure
$ make
```

You can also build groff in an out-of-source build tree, which is cleaner:

```
$ mkdir build
$ cd build
$ ../configure
$ make
```

Parallel build is also supported: ‘make’ can be invoked with the -j option, which will greatly speed up the build.

1.2. Automake in the autotools process

Automake’s main job is to generate a Makefile.in file (this file is maintained manually on projects using only autoconf). The main file processed by ‘automake’ is the Makefile.am file, which eventually generates a Makefile. The (simplified) process is:

- ‘aclocal’ generates the ‘aclocal.m4’ file from ‘configure.ac’ and the user-defined macros in ‘acinclude.m4’.
- ‘autoheader’ generates config.h.in.
- ‘autoconf’ generates the ‘configure’ script from ‘aclocal.m4’ and ‘configure.ac’
- ‘automake’ generates Makefile.in from Makefile.am and the ‘configure.ac’ file. It also generates some helper scripts, on the groff project they are located in build-aux.
- ‘configure’ generates ‘config.status’
- ‘config.status’ generates the Makefile and config.h.

Finally, ‘autoreconf’ is the program that can be used to call these various tools in the correct order.

Automake defines a set of special variables that are used to generate various build rules in the final Makefile. Note however that if Automake’s predefined rules are not enough, you still have the possibility of adding handwritten standard ‘make’ rules in a Makefile.am; these rules will be copied verbatim in the Makefile.in and then in the final Makefile.

1.3. Modification of autotools files

Previously, when groff used ‘autoconf’ only and not ‘automake’, you had to invoke manually the autotools, depending on what you modified. For example, to change the file ‘aclocal.m4’, you had to run the shell command ‘aclocal -I m4’; to recreate the files ‘configure’ and ‘Makefile’, you had to use the command ’autoreconf - I m4’.

Now, as groff uses ‘automake’, you don’t need to run ‘autoreconf’. If you make some changes in Makefile.am or configure.ac, all the files that need to be updated will be

regenerated when you execute ‘make’.

2. Building a program

2.1. A program and its source files

Generally speaking, when using ‘automake’ you will have to write a Makefile.am file and use the variable **bin_PROGRAMS** to declare a program that should be built, and then list the sources of this program in a variable that starts with the name of your program and ends with **_SOURCES**. In the groff project we have only 1 top-level Makefile.am that includes several .am files.

Take for example the build of grolbp, in src/devices/grolbp/grolbp.am. The file starts with:

```
bin_PROGRAMS += grolbp
```

This says that a program named ‘grolbp’ is added to the list of the programs that should be built. The variable **bin_PROGRAMS** is initialized to an empty string in the top-level Makefile.am, which includes grolbp.am. (We will see later why we don’t write directly **bin_PROGRAMS = grolbp** in a Makefile.am in the grolbp directory.)

Then, we list the sources of grolbp like this:

```
grolbp_SOURCES = \
  src/devices/grolbp/lbp.cpp \
  src/devices/grolbp/lbp.h \
  src/devices/grolbp/charset.h
```

As you added ‘grolbp’ to **bin_PROGRAMS**, you need to define the sources of grolbp in the variable **grolbp_SOURCES**. If you write in another file **bin_PROGRAMS += foo** you will list the sources of ‘foo’ in **foo_SOURCES**.

With these two statements, the resulting generated Makefile will contain everything that is needed to build, clean, install and uninstall the ‘grolbp’ binary when invoking the adequate ‘make’ command. Also, the source files listed in**grolbp_SOURCES** will automatically be included in the distribution tarball. That is why the headers are also listed in **grolbp_SOURCES**: it is not necessary to add them in order to correctly build ‘grolbp’, but this way the headers will be distributed.

- The path to the files are relative to the top-level directory.
- The binaries are generated in the top-level build directory.
- The .o files are generated in the directory where the source files are located, or, in the case of an out-of-source build tree, in a directory that is the replication of the source tree directory. For example if you built groff in a ‘build’ directory, lbp.o (object file from src/devices/grolbp/lbp.cpp) will be located in build/src/devices/grolbp/lbp.o.

We will also see later the reasons; this is due to the non-recursive make design.

2.2. Linking against a library

To list which libraries grolbp needs to link against, we just write:

```
grolbp_LDADD = $(LIBM) \
  libdriver.a \
  libgroff.a \
  lib/libgnu.a
```

Again, we use the variable **grolbp_LDADD** because we added a program named ‘grolbp’. This will also automatically set build dependencies between ‘grolbp’ and the libraries it needs: ‘libdriver.a’ and ‘libgroff.a’, that are convenience libraries built within the groff project, will be compiled before grolbp.

2.3. Preprocessor flags

Preprocessor flags that are common to all the binaries are listed in the variable **AM_CPPFLAGS** in the top-level Makefile.am. If a ‘foo’ binary needs specific preprocessor flags, use **foo_CPPFLAGS**, for example, in src/devices/xditview/xditview.am, extra flags are needed to build gxditview and are added like this:

```
gxditview_CPPFLAGS = $(AM_CPPFLAGS) $(X_CFLAGS) -Dlint \
  -I$(top_builddir)/src/devices/xditview
```

The use of specific CPPFLAGS changes the name of the generated objects: the .o object files are prefixed with the name of the program. For example, the .o file corresponding to src/devices/xditview/device.c will be src/devices/xditview/gxditview-device.o.

2.4. Cleaning

You don’t need to write rules to clean the programs listed in **bin_PROGRAMS**, ‘automake’ will write them for you. However, some programs might have generated sources that should be cleaned. In this case, you have mainly two special variables to list extra files that should be cleaned:

- **MOSTLYCLEANFILES** for files that should be cleaned by ‘make mostlyclean’
- **CLEANFILES** for files that should be cleaned by ‘make clean’

There is also the possibility of writing custom rules. We will see that later.

2.5. Dependencies

We have already seen that when linking against a convenience library, the dependencies are already created by ‘automake’. However, some dependencies still need to be manually added, for example when a source file includes a generated header. In this case, the easiest way is to

add a plain-make dependency. For example, src/roff/groff/groff.cpp includes defs.h, which is a generated header. We just add in src/roff/groff/groff.am:

```
src/roff/groff/groff.$(OBJEXT): defs.h
```

2.6. Scripts

Apart from **bin_PROGRAMS**, there is another similar special variable for scripts: **bin_SCRIPTS**. The scripts listed in this variable will automatically be built (of course you have to provide your custom rule to build the script), installed and uninstalled when invoking ‘make’, ‘make install’ and ‘make uninstall’. The main difference is that unlike the programs listed in **bin_PROGRAMS**, the scripts will not be cleaned by default. They are not distributed by default either. In the groff project, **bin_SCRIPTS** are cleaned because they are added to **MOSTLYCLEANFILES** in the top-level Makefile.am.

A simple example are the gropdf and pdfmom scripts in src/devices/gropdf/gropdf.am:

```
bin_SCRIPTS += gropdf pdfmom
  [...]
gropdf: $(gropdf_dir)/gropdf.pl $(SH_DEPS_SED_SCRIPT)
        $(AM_V_GEN)$(RM) $@ \
        sed -f $(SH_DEPS_SED_SCRIPT) \
            -e "s|[@]VERSION[@]|$(VERSION)|" \
            -e "s|[@]PERL[@]|$(PERL)|" \
            -e "s|[@]GROFF_FONT_DIR[@]|$(fontpath)|" \
            -e "s|[@]RT_SEP[@]|$(RT_SEP)|" $(gropdf_dir)/gropdf.pl \
            >$@         && chmod +x $@

pdfmom: $(gropdf_dir)/pdfmom.pl $(SH_DEPS_SED_SCRIPT)
        $(AM_V_GEN)$(RM) $@ \
        sed -f $(SH_DEPS_SED_SCRIPT) \
            -e "s|[@]VERSION[@]|$(VERSION)|" \
            -e "s|[@]RT_SEP[@]|$(RT_SEP)|" \
            -e "s|[@]PERL[@]|$(PERL)|" $(gropdf_dir)/pdfmom.pl \
            >$@
        && chmod +x $@
```

In this example, the ‘@' symbol is protected by square brackets to prevent the substitution of the variable by ‘automake'.

3. Non-recursive make schema

There are two possibilities for organizing the Makefile.am of a large project, using a recursive or a non-recursive ‘make'.

3.1. 1st possibility: make recursion

A top level Makefile.am includes another Makefile.am, using the **SUBDIRS** directive, and the Makefile.am of each sub-directory lists the programs that should be built. If we had chosen this type of organization, we would have a Makefile.am in src/devices/grolbp and in each directory that contain sources to build a program (tbl, eqn, troff, and so on). We would write in the top-level Makefile.am:

```
SUBDIRS = src/devices/grolbp \
  ... (and all the dir that build a program or a script)
```

and in src/devices/grolbp, we would have a file Makefile.am that contains:

```
bin_PROGRAMS = grolbp
grolbp_SOURCES = lbp.cpp lbp.h charset.h
```

Only ‘grolbp’ is affected to the variable **bin_PROGRAMS**. It would be the same in, say, src/roff/troff: you would have a Makefile.am with **bin_PROGRAMS = troff**. We would have one generated Makefile per Makefile.am file: in the build tree you will have the top-level Makefile, grolbp’s Makefile in src/devices/grolbp, troff’s Makefile in src/roff/troff, and so on. When calling ‘make’ to build everything, ‘make’ will be recursively called in all the directories that have a Makefile. Thus, the paths are logically relative to the directory that contains the Makefile.am.

This approach has the disadvantage of making dependencies harder to resolve: each Makefile does not know the targets of the other Makfiles. It also makes the build slower.

3.2. Non-recursive make used by the Groff project

The second possibility, which was chosen for the groff project, is to use a non-recursive make schema. It is described in paragraph 7.3 of the Automake manual (“An Alternative Approach to Subdirectories”), based on the following paper from Peter Miller: *Recursive Make Considered Harmful*.

The idea is to have a single Makefile that contains all the rules. That is why we have only a single Makefile.am in the top-level directory which includes all the .am files that define rules to build the various programs. The inclusion is done with the **include** directive, not **SUBDIRS**. Using ‘include’ is like copying the contents of the included file into the top-level Makefile.am, and will not generate other Makefile.

We first say in this top-level Makefile.am:

```
bin_PROGAMS =
```

and then all the .am files that define a program to be built (e.g. src/devices/grolbp/grolbp.am, src/roff/troff/troff.am, and so on) overload this variable, so that at the end, all the programs that should be built are listed in this **bin_PROGRAMS** variable. This is the reason why all the paths in the various .am files are relative to the top-level directory: at the end we will have

only one Makefile in the top-level directory of the build tree.

As the resulting single Makefile knows all the targets, the dependencies are easier to manage. The build is also faster, particularly when compiling a single file: ‘make’ is called once only and the file will be instantly rebuilt, while on a recursive make system, ‘make’ will have to be invoked in all the sub-directories.

Note also that in order to make ‘gnulib’ work with this non-recursive schema, the ‘--automake-subdir’ configuration should be selected in bootstrap.conf.

4. Installing data

Variables that end with **_DATA** are special variables used to list files that should be installed in a particular location. The prefix of the variables should refer to another previously defined variable that ends with a ‘dir’ suffix. This variable that ends with ‘dir’ defines where the files should be installed.

4.1. A simple case

For example, in font/devX100/devX100.am, we can see this:

```
if !WITHOUT_X11
devX100fontdir = $(fontdir)/devX100
devX100font_DATA = $(DEVX100FONTS)
endif

EXTRA_DIST += $(DEVX100FONTS)
```

DEVX100FONTS is just a list of font files, defined at the beginning of devX100.am. **fontdir** is where all the font directories are installed, it is defined in the top-level Makefile.am. The conditional **if !WITHOUT_X11** is used to prevent the installation of these files if X11 is not available.

We first define where we wants to install the devX100 fonts with:

```
devX100fontdir = $(fontdir)/devX100
```

Because we declared a variable ending with ‘dir’, we are allowed to define **devX100font_DATA** (you remove the ‘dir’ suffix and add **_DATA**). Wildcards are not supported in the special variables that end with **_DATA**.

With these two lines, ‘make install’ will install the files listed in **DEVX100FONTS** and ‘make uninstall’ will uninstall them. **devX100fontdir** will be automatically created if missing during the installation process, but not removed during the uninstall. The complete **fontdir** is removed by a custom uninstall rule (uninstall_groffdirs in Makefile.am).

Because the files listed in **devX100font_DATA** are not distributed by default, we explicitly added them to the **EXTRA_DIST** variable, which lists all the files that should be distributed and that are not taken into account by the default automake rules.

```
EXTRA_DIST += $(DEVX100FONTS)
```

Another possibility would have been to add a ‘dist’ prefix to the **devX100font_DATA** variable, in this case the use of **EXTRA_DIST** is useless (except of course if **WITHOUT_X11** is true, in this case we don’t install the files but we still have to distribute them):

```
if !WITHOUT_X11
devX100fontdir = $(fontdir)/devX100
dist_devX100font_DATA = $(DEVX100FONTS)
else
EXTRA_DIST += $(DEVX100FONTS)
endif
```

4.2. Dealing with generated files

In the previous example, all the font files that must be installed were already present in the source tree. But in some cases, you need to generate the files you intend to install. In this case, the files should be installed but not distributed. A simple way to deal with this is to add a ‘nodist’ prefix to your **xxx_DATA** variable.

For example in font/devps/devps.am, we have a list of font files already present in the source tree, defined by **DEVPSFONTFILES**, and another list of font files that are generated, listed in the variable **DEVPSFONTFILES_GENERATED**. They should all by installed in a ‘devps’ directory under the fontdir. Thus the following three lines, where we use the ‘dist’ and ‘nodist’ prefixes:

```
devpsfontdir = $(fontdir)/devps
dist_devpsfont_DATA = $(DEVPSFONTFILES)
nodist_devpsfont_DATA = $(DEVPSFONTFILES_GENERATED)
```

The generated files are not cleaned by default, thus we add:

```
MOSTLYCLEANFILES += $(DEVPSFONTFILES_GENERATED)
```

5. Extending Automake’s rules

5.1. Local clean rules

In most of the cases, the files that need to be cleaned are automatically determined by ‘automake’, or were added to the **MOSTCLEANFILES** or **CLEANFILES** variables. However, you might need to define a specific rule to clean some files that were not added to any list. Automake defines a set of targets to extend the clean targets with your own rules: clean-local, mostlyclean-local, distclean-local or maintainerclean-local. An example of such extension exists in font/devpdf/devpdf.am: because some fonts are not explicitly listed in a **xxx_DATA**

variable but generated by a custom rule, we define an extra rule to extend the ‘mostlyclean’ target:

```
mostlyclean-local: mostlyclean_devpdf_extra
mostlyclean_devpdf_extra:
        @echo Cleaning font/devpdf
        rm -rf $(top_builddir)/font/devpdf/enc \
          $(top_builddir)/font/devpdf/map;
        if test -d $(top_builddir)/font/devpdf; then \
          for f in $(GROFF_FONT_FILES); do \
            rm -f $(top_builddir)/font/devpdf/$$f; \
          done; \
        fi
```

5.2. Local install/uninstall rules and hooks

Similarly to the clean rules, there are extensions to install and uninstall rules. They come with two flavous, local rules and hooks.

- There are 2 rules to extend install commands: ‘install-exec-local’ for binaries and ‘install-data-local’ for data.
- There is 1 uninstall local rule: ‘uninstall-local’.

There are no guarantees on the order of execution of these local rules. An example of local rule is the installation of GXditview.ad and GXditview-color.ad files in src/devices/xditview/xditview.am: if theses files are already installed, the old files are first saved. Also, the final file that is installed is stripped from its .ad suffix. Thus the usage of a custom rule rather than the definition of a **`xxx_DATA`** variable:

```
# Custom installation of GXditview.ad and GXditview-color.ad
install-data-local: install_xditview
uninstall-local: uninstall_xditview

[...]
install_xditview: $(xditview_srcdir)/GXditview.ad
        -test -d $(DESTDIR)$(appdefdir) \
          || $(mkinstalldirs) $(DESTDIR)$(appdefdir)
        if test -f $(DESTDIR)$(appdefdir)/GXditview; then \
          mv $(DESTDIR)$(appdefdir)/GXditview \
            $(DESTDIR)$(appdefdir)/GXditview.old; \
        fi
        [...]
        $(INSTALL_DATA) $(xditview_srcdir)/GXditview.ad \
```

```
$(DESTDIR)$(appdefdir)/GXditview
```

Hooks, on the other hand, are guaranteed to be executed after all the standard targets have been executed.

- There are 2 install hooks: ‘install-exec-hook’ and ‘install-data-hook’.
- There is 1 uninstall hook: ‘unintall-hook’

An example of hook is the ‘uninstall_groffdirs’ rule in the top-level Makefile.am. This hook is used to remove all the directories specific to groff introduced by the installation process. Obviously it could not be a local extension of ‘uninstall’ because the order of execution is not guaranteed.

```
# directories specific to groff
uninstall-hook: uninstall_groffdirs
uninstall_groffdirs:
        if test -d $(DESTDIR)$(datasubdir); then \
          rm -rf $(DESTDIR)$(fontdir); \
          rm -rf $(DESTDIR)$(oldfontdir); \
          rmdir $(DESTDIR)$(datasubdir); \
        fi
        [...]
```