Source code trees IN THE VALLEY OF THE CODE

THORSTEN FISCHER

With the help of Automake and Autoconf, you can create easily installed source code text trees. Read on to find out how.

So you've just written yet another terrific GNOME program. Great! But does it, like so many other great programs, lack something in terms of ease of installation? Even the best and easiest to use programs will cause headaches if you have to type in lines like this,

```
gcc -c sourcee.c gnome-config –libs –cflags
gnome gnomeui gnomecanvaspixbuf -o sourcee.o
```

perhaps repeated for each of the files, and maybe with additional compiler flags too, only to then demand that everything is linked. And at the end, do you then also have to copy the finished binary manually into the destination directory? Instead, wouldn't you rather have an easy, portable and quick installation process? Well, you can – if you know how.

Front end

The complicated installation scenario we described above is of course a bit of an exaggeration, since Makefiles are not hard to write. Using these, all you need to do is type *make* in the source text directory, and the program is created. A "simple" Makefile for a short C-program in GNOME can look something like the one in Listing 1. Not too complex, eh? Unfortunately, creating a Makefile isn't always the best solution, as assumptions on programs locations, path names and others things may not be true in all cases, forcing the user to edit the file in order to get it to work properly.

Listing 1: A simple Makefile for a GNOME

```
1: CC=/usr/bin/gcc
2: CFLAGS=`gnome-config –cflags gnome gnomeui`
3: LDFLAGS=`gnome-config –libs gnome gnomeui`
4: OBJ=example.o one.o two.o
5: BINARIES=example
6:
7: all: $(BINARIES)
8:
9: example: $(OBJ)
10:      $(CC) $(LDFLAGS) -o $@ $(OBJ)
11:
12: .c.o:
13:      $(CC) $(CFLAGS) -c $<
14:
15: clean:
16:      rm -rf $(OBJ) $(BINARIES)
```

That's not what you wanted, is it? No, what you wanted is something much simpler. Something like this, perhaps?

```
./configure
make
make install
```

But hang on, we're getting ahead of ourselves. First we need to deal with the source text tree and the

GNOME-specific properties that must be taken into account when building one.

Structure

It is important to have a structure for the source text tree. The source code itself should be located separately from other things such as the documentation or the files for configuration – doing so makes it easier to get an overview. So the first thing we'll do is to create a directory called *example*, which will be the site for our tree, then create a *src* within it for our code and throw everything we need in there. To stick with the files used in the Makefile example in Listing 1, this means the files *example.c*, *one.c*, *one.h*, *two.c* and *two.h*. The first of these files is shown in Listing 2, while the other four are empty and are only included as an example.

Listing 2: example.c

```
1: #include <gnome.h>
2: #include "one.h"
3: #include "two.h"
4:
5: int main (int argc, char *argv [])
6: {
7:       GtkWidget *app;
8:
9:       gnome_init ("example", "0.0.1", arg➥
c, argv);
10:      app = gnome_app_new ("example", "Exa➥
mple");
11:
12:      gtk_widget_show_all (app);
13:      gtk_main ();
14:      return TRUE;
15: }
```

Documentation: A tiresome step for every programmer, but one that co-developers and users will be grateful for. The following files are the done thing to put in a source directory:

```
* Authors: The authors are listed here
* ReadMe: Everything worth reading on th➥
e program
* News: News concerning the program
* ChangeLog: Documentation of all changes
* Copying: A copy of the GNU GPL
* Install: Installation instructions
```

These are the most important files, but there are others too. Where do these files come from? The first three are obviously ones you have to make yourself, while the last two should preferably be copied from the *automake* directory.

Automake

automake and *autoconf* are two small GNU tools, which can be obtained from *ftp.cs.tu-berlin.de/pub/gnu/*. These tools create your configuration files for you, which then only have to be executed by the user in order to get everything done for them. Users don't have to have installed the aforementioned programs themselves, mind you, as the source text package from our application is all that is needed for this procedure. How the two programs should be installed (if you can't simply take them from the developer section of your distribution CD that is) is something you can probably guess at.

But before we attach *automake* and *autoconf* to our sources, there is still some preparatory work to be done. We need to create two more files – namely *configure.in* and *Makefile.am* – examples of which can be seen in Listings 3 and 4.

Listing 3: configure.in

```
1: AC_INIT(src/example.c)
2:
3: AM_CONFIG_HEADER(config.h)
4:
5: AM_INIT_AUTOMAKE(Example, 0.1.0)
6:
7: AM_MAINTAINER_MODE
8:
9: AM_ACLOCAL_INCLUDE(macros)
10:
11: GNOME_INIT
12:
13: AC_PROG_CC
14: AC_ISC_POSIX
15: AC_HEADER_STDC
16: AC_ARG_PROGRAM
17: AM_PROG_LIBTOOL
18:
19: GNOME_COMPILE_WARNINGS
20:
21: ALL_LINGUAS="de"
22: AM_GNU_GETTEXT
23:
24: AC_SUBST(CFLAGS)
25: AC_SUBST(CPPFLAGS)
26: AC_SUBST(LDFLAGS)
27:
28: AC_OUTPUT(
29: Makefile,
30: macros/Makefile,
31: src/Makefile,
32: intl/Makefile,
33: po/Makefile.in
34: )
```

Listing 4: Makefile.am

```
1: SUBDIRS=macros po intl src
2:
3: Applicationsdir=$(datadir)/gnome/apps/Ap➥
plications
4: Applications_DATA=example.desktop
```

The two files are basically easy to explain. We'll start with *configure.in*.

In the first line *autoconf* is initialised, with the name in brackets of any existing file. The main source file is ideal for this. *AM_CONFIG_HEADER* specifies a header file, which is intended later to carry the specific information for the configured package and which must also be integrated into the source text – but more on that later. Note the different prefixes that the macro names carry: *AC_* designates a macro for *autoconf*, and *AM_* (hardly surprisingly) refers to *automake*. *automake* will also

be concerned with the content of this file, but again more on that later. It will be initialised in line 5 complete with the name of the package and its version number. If, during the course of development, you feel enough has happened to push this number up a notch, then don't forget to note the fact here as well as elsewhere.

In the ninth line, the *macros* directory is added to the *aclocal* search path. This is yet another thing we haven't mentioned yet, and it won't be the last. It deals with the administration of the macros which are called up in *configure.in* (*macros* can be copied from the *gnome-libs* source package, by the way). This is then followed by the initialisation of GNOME, various standard macros to search and test a C-compiler, some header files, POSIX-conformity of the system and so on. Then in line 19 compiler warnings are switched on.

Line 21 is concerned with localisations of the program; our example assumes the existence of "de" translations, but any supported locale can be used. The next line, *gettext*, is also necessary for localisation. You don't have to create international installations, of course, but it really does add a touch of professionalism – and is also a great way to show off your language skills.

The lines 24, 25 and 26 export the variables, compiler and linker flags, which have been defined during the processing of the file, so that they can actually be used in the program. The last lines following *AC_OUTPUT* finally specify where Makefiles should be created. Line 33 is not a typo, by the way.

Makefile.am

This is a template file, to create – via an intermediate step when it will be called *Makefile.in* – an individual or all completed Makefiles respectively. The first line lists all subdirectories in which additional templates are located and/or in which Makefiles should be created. Lines three and four specify the directory in which our program should place its Desktop file, with the aid of which it will later pop up in the GNOME menus.

One more item of importance now is the *Makefile.am* in the subdirectory *src*, containing the actual sources of the program. An example can be seen in Listing 5.
I have adapted this example from Havoc Pennington's 'Gtk+/GNOME Application Development'. Firstly, the Include-Directories are defined, and then the source files for the finished program are named. Finally the flags for the linker are set, which should amalgamate the compiled object files.

Et voila

And now it's almost done! The following commands now deal with the creation of our configuration scripts, Makefiles and so on:

```
frog@verlaine:~/code/example # libtoolize ➥
--copy --force
frog@verlaine:~/code/example # gettextize –c➥
opy –force
frog@verlaine:~/code/example # aclocal
frog@verlaine:~/code/example # autoheader
frog@verlaine:~/code/example # automake –ad➥
d-missing –copy
frog@verlaine:~/code/example # autoconf
```

The first command is necessary mainly for creating libraries. It also copies scripts into the directory, which are needed elsewhere. —*copy* requests copying rather than the creation of Symlinks – the normal default setting – and —*force* creates the files again, even when they already exist. *gettextize* gives the package the necessary files for internationalisation and localisation. *aclocal* edits the macros and *autoheader* makes a file *config.h.in*, which is then created by *automake* and *autoconf*. Now, for package creation, we just have the easy target *dist*: after calling up *configure* a *make dist* produces a ready-wrapped parcel, in our example called *example-0.1.0.tar.gz*. In the *macros* directory, you'll find a little script called *autogen.sh*, which can take over these calls for you. You don't have to keep executing these by hand once you have added a source text file.

Listing 5: src/Makefile.am

```
1: INCLUDES=$(top_srcdir) -I$(includedir)$➥
(GNOME_INCLUDEDIR) \
2: -DG_LOG_DOMAIN=\"Example\" \
3: -DGNOMELOCALEDIR=\""$(datadir)/locale"\" \
4: -I../intl -I$(top_srcdir)/intl
5:
6: bin_PROGRAMS=example
7:
8: example_SOURCES=example.c \
9:       one.h \
10:      two.h \
11:      one.c \
12:      two.c
13:
14: example_LDADD=$(GNOMEUILIBS) $(GNOME_LIB➥
DIR) $(INTLLIBS)
```

Listing 6: src/example.c

```
1: #include <gnome.h>
2: #include <config.h>
3: #include "one.h"
4: #include "two.h"
5: int main (int argc, char *argv [])
6: {
7: GtkWidget *app;
8:
9: gnome_init (PACKAGE, VERSION, argc, argv);
10: app = gnome_app_new (PACKAGE, _("Example➥
"));
11:
12: gtk_widget_show_all (app);
13: gtk_main ();
14: return TRUE;
15: }
```

Changes in the sources

As the result of the creation of the file *config.h* there may be some other changes to the source text. In particular details of the name of the package and the version number can now be accessed more easily. Listing 6 shows the code for the example file after the changes. The macro *_()* in the tenth line is needed because of our desire to internationalise our package.

The desktop file

In our main source directory, the empty file *example.desktop* will still be lurking around. If this is filled with content, as can be seen from Listing 7, and if the line

```
EXTRA_DIST = example.desktop
```

is entered in its main *Makefile.am*, then when the completed program is installed a Desktop entry will be added to the GNOME menu hierarchy. This entry may contain localised names – such as in German as shown here–, a comment – again localised –, the name of the executed file, the file type and the fact that the application should not be executed in a terminal.

Listing 7: example.desktop

```
1: [Desktop Entry]
2: Name=Example program
3: Name[de]=Beispielprogramm
4: Comment=An example
5: Comment[de]=Ein Beispiel
6: Exec=example
7: Terminal=0
8: Type=Application
```

Glade

I shouldn't really be telling you this right at the end, but a program called Glade (see *glade.pn.org/*) can create complete source text trees for you at the push of a button. This doesn't mean you shouldn't bother working "by hand", as described in this article, though. Why? Well, primarily because Glade can only create a quite rudimentary tree. As soon as you want more than Glade has to offer, you have to get to work by hand anyway, and this will only makes sense to you if you have some prior knowledge of doing so, which we've just given you. Indeed, having created your own source tree, you can sleep soundly at night in the knowledge that if there is ever any problem then you can make your own changes in no time, without having to throw yourself on the mercy of a graphical user interface.

So there you have it – how to create easily installed packages in a nutshell. Do please give it a try – you'll be making the Linux world a better place for everybody if you do. ■