LINUX VOICE TUTORIAL

LINUX 101: COMPILING SOFTWARE FROM SOURCE CODE

MIKE SAUNDERS

Binary packages are all good and well, but to get the latest features and useful patches, it's worth building programs from source.

WHY DO THIS?

- Get the latest programs without waiting for your distro to package them up for you
- Enable non-standard features and add new ones with patches
- Stay extra secure by using binaries that you've compiled yourself

You might think that it's utterly pointless to compile programs from their original, human-readable source code, given how many awesome binary package managers exist in the Linux world. And fair enough: in most cases it's better to grab something with **apt-get** or **yum** (or whatever your distribution uses) than to take the extra steps required to build things by hand. If a program is in your distro's repositories, is up-to-date and has all the features you need, then great – enjoy it.

But it doesn't always work like that. Most distros aren't rolling-releases (Arch is one of the few Linux distributions that is) so you only get new versions of packages in the stable branches once or twice a year. You might see that FooApp, one of your favourite programs, has just reached version 2.0, but only version 1.5 is available in your distro's package repositories. So what do you do? If you're in luck, you might find a third-party repository for your current distro release with the latest version of FooApp, but otherwise you need to compile the new release from its source code.

And that's not the only reason to do it: you can often enable hidden, experimental or unfinished features by building from source. On top of this, you can apply patches from other developers that add new features or fix bugs that the software's maintainers haven't yet sorted out. And when it comes to security, it's good to know that the binary executables on your machine have been generated from the original developer's source code, and haven't been tampered with by a malicious distro developer. (OK, this isn't a big worry in Linux, but it's another potential benefit.)

We've had several requests to run a tutorial on compiling software, and explain some of the black magic behind it. We often talk about compiling programs in our FOSSpicks section, as it's the only way to get them – so if you've had trouble in the past, hopefully you'll be fully adept after reading the next few pages. Let's get started!

1 GRABBING THE SOURCE

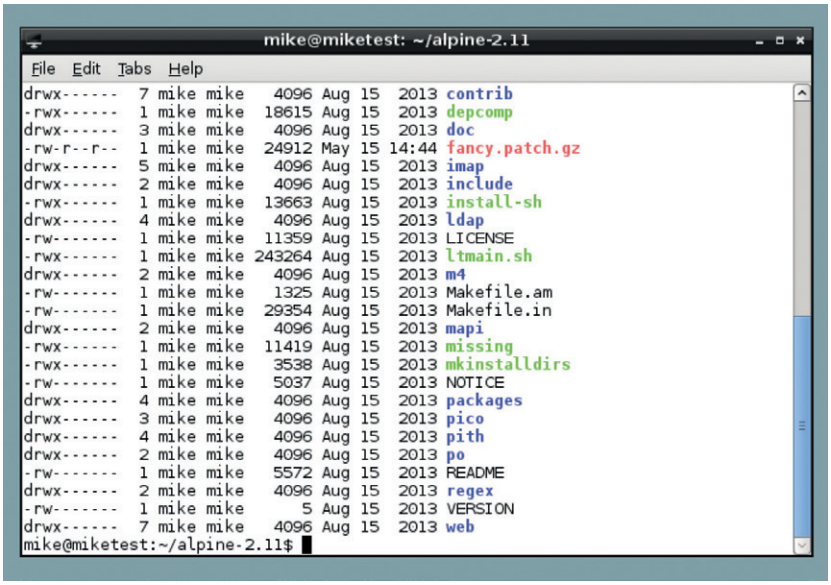
Although there are various build systems in use in the Free Software world, we'll start with the most common one, generated by a package called GNU Autotools. Compiling software makes heavy use of the command line – if you're fairly new to Linux, check out the Command Line Essentials box on the facing page before you get started here, so that you don't get lost as soon as you start.

Here we're going to build Alpine, a (very good) text-mode email client that works as an ideal example for this tutorial. We'll be using Debian 7 here, but the process will be similar or identical across other distributions as well. These are the steps we're going to take, and you'll use all or most of them with other programs you compile:

1. Download the source code and extract it.
2. Check out the documentation.
3. Apply patches.
4. Configure to your liking.
5. Compile the code.
6. Install the binary executable files.

Alpine is a continuation of the Pine email client of yesteryear. If you search for its official site you'll see that the "latest" version is 2.00, but that's ancient – some developers have continued hacking away on it elsewhere, so go to **http://patches.freeiz.com/alpine** to get version 2.11. (If a newer version has arrived by the time you read this article, adjust the version numbers in the following commands accordingly.) The source code is contained in **alpine-2.11.tar.xz**, so open a terminal and grab it like

Normally, documentation files are in all uppercase and contain plain text – eg **LICENSE**, **README** and **VERSION** here.



so:

```
wget http://patches.freeiz.com/alpine/release/src/alpine-2.11.tar.xz
```

This is a compressed archive that needs to be extracted. You can use the same command for archives that end in **.tar.gz** and **.tar.bz2**:

```
tar xfv alpine-2.11.tar.xz
```

(If the file ends in **.zip**, try **unzip <filename>**.) As the archive is extracted, you'll see a bunch of files whizz by on the screen – enter **ls** when the process is done and you'll see a new directory. Enter **cd alpine-2.11** to switch into it. Now enter **ls** again to have a nosey around and see what's inside.

If you see a green file called **configure**, that's great – you can almost certainly start building the software straight away. But nonetheless, it's wise to check out the program's own documentation first. Many applications include **INSTALL** and **README** files along with the source code; these are plain text files that you can read with the **less** command. Sometimes the **INSTALL** file will contain "generic installation instructions", with hundreds of lines of boring, non-app-specific information, so it's a good idea to ignore it. If the **INSTALL** file is short, to-the-point and written specifically for the program in hand, skim through it to see what you need to do.

Command line essentials

If you're new to Linux and the command line, here are some super quick tips. Open a command line via Terminal, Konsole or XTerm in your desktop menu. The most useful commands are **ls** (to list files, with directories shown in blue); **cd** (change directory, eg **cd foo/bar/**, or **cd ..** to go down a directory); **rm** (remove a file; use **rm -r** for directories); **mv** (move/rename, eg **mv foo.txt bar.txt**), and **pwd** (show current directory).

Use **less file.txt** to view a text file, and **q** to quit the viewer. Each command has a manual page (eg **man ls**) showing options – so you can learn that **ls -la** shows a detailed list of files in the current directory. Use the up and down arrow keys to cycle back through previous commands, and use Tab to complete file or directory names: eg if you have **longfilename.txt**, enter **rm long** and then hit Tab should complete the filename.

Similarly, check out the **README** as well. Alpine only has a **README** file, but it's fairly decent, explaining the commands required to build the source code, and listing the binary executable files that will be produced. It's lacking something important, though: a list of dependencies. Very few programs can be compiled with just a standalone compiler, and most will need other packages or libraries installed. We'll come to this in a moment.

LV PRO TIP

If you want to uninstall a program you've compiled from source, you can usually run **make uninstall** (as root) at the same stage that you'd run **make install** in the text. This removes the files that the command put in place earlier.

2 APPLYING PATCHES

So, we've done steps 1 and 2 – downloading the source and reading the documentation. In most cases you'd want to go straight on the compilation step, but occasionally you may prefer to add a patch or two beforehand. Put simply, a patch (aka a "diff") is a text file that updates lines in the source code to add a new feature or fix a bug. Patches are usually very small in comparison to the original code, so they're an efficient way to store and distribute changes.

If you go back to **http://patches.freeiz.com/alpine**, you'll see a list of "Most popular patches" near the top. Click on the "Enhanced Fancy Thread Interface" link to go to **http://patches.freeiz.com/alpine/info/fancy.html**. Along the top you'll see links to patches for various Alpine versions – so because we have Alpine 2.11, click the link with that number to download **fancy.patch.gz**.

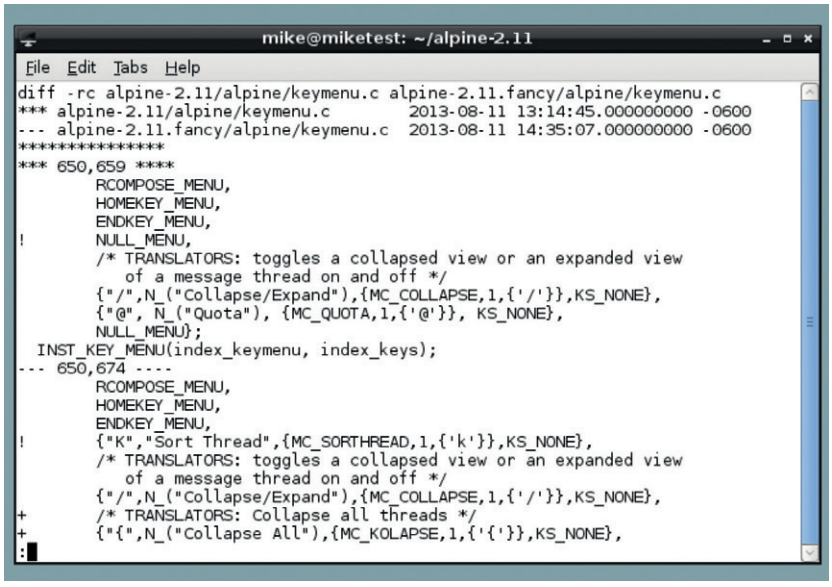
Now move that file into your **alpine-2.11/** directory. You might be curious to have a peek inside the patch to see how it works, but as it's compressed you'll need to enter:

```
zless fancy.patch.gz
```

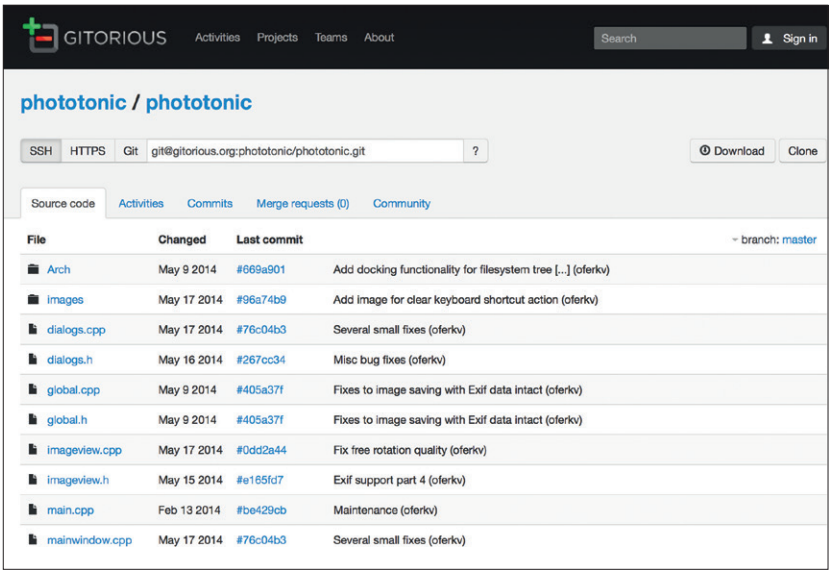
Lines starting with ******* show which source code files are to be modified, and the **+** and **!** characters at the start of a line show lines that should be added or changed respectively. So if you see something that looks like this:

```
  char *debug_str = NULL;
  char *sort = NULL;
+ char *threadsort = NULL;
```

This means that the new "threadsort" line in the patch should be added after the first two (which already exist in the original code). But why doesn't the patch simply use line numbers? Well, it's possible to do it that way, but then the patch becomes useless if you make even the tiniest change to the file yourself. If you add a line, all of the line numbers in the patch become out of sync, so you need to make a new one. By having a few lines from the original code in the patch, you have some context, so the patch can normally still be applied even if the code has been



An example of a typical patch: changed lines of code begin with a **!** symbol, whereas new lines have **+** at the start.



Many open source programs and games, such as those we cover in FOSSpicks, are only provided as source code.

changed by a different patch. To apply a patch, you need to use the (surprise!) **patch** command.

This is installed by default in most distributions, so you shouldn't need to go hunting for it anywhere. You can test the effects of the patch without actually having to make any changes to the code by using the **--dry-run** option, like so:

```
zcat fancy.patch.gz | patch -p1 --dry-run
```

Here, **zcat** extracts the patch into plain text, and then it's piped with the **|** character into the patch tool. (If you've never used it before, the pipe character is a massively useful way to move data between programs – you can send the output of one program straight to another, without redirecting it via text files).

Anyway, we use **-p1** in this **patch** command because we're already inside the source code directory; if you're outside it (like, a level above in the filesystem) or the patch doesn't work, try removing it. Once you execute this command, you'll see lines like:

```
patching file alpine/setup.c
```

If it all works, re-run the command omitting the **--dry-run** option, and the changes will be made permanent. Congratulations – you've just spruced up Alpine with a new feature! Some programs have hundreds of patches from other developers, and patches are often rolled into the main source code once they've been well tested.

3 CONFIGURING AND COMPILING

We're almost ready to compile the source code, but there's still one more important step: configuration. As mentioned earlier, many programs have features and experimental options that are not compiled into the executables by default, but can be enabled by advanced users. (Why don't the developers simply include the features, but have a command line switch to enable them? Well, certain features can impact the stability of the overall code, so they're not compiled in by default until they're deemed as reliable.)

Enter the following command:

```
./configure --help | less
```

This runs the **configure** script in the current directory and spits out its help text to the **less** viewer (that's a pipe symbol before the **less** command). Scroll down and you'll see that there's a huge list of options you can change: the installation prefix (where it should be installed, eg **/usr/local/**), where the manual pages should go, and so forth. These are pretty generic and apply to almost every program you build from source using this process, so scroll down to the Optional Features section – this is where the fun begins.

In this section you'll find features specific to Alpine. The Optional Packages section beneath it also has things that you can enable or modify using the relevant command line flags. For instance, if you have a command line spellchecking program that you love, and want to use it inside Alpine, you'll see that there's a **--with-interactive-spellcheck** option. You would use it like so:

```
./configure --with-interactive-spellcheck=progname
```

Providing **progname** is in your usual paths (eg **/bin**, **/usr/bin**, **/usr/local/bin**) then this should work.

Many of the options in Alpine's configure script let you disable things rather than enable them. This is

The CMake alternative

While many programs still use the GNU Autotools approach of **./configure**, **make** and **make install** (especially those programs that are part of the GNU project), an alternative is becoming increasingly popular: CMake. This does a similar job, but it requires different commands, so we'll go through them here. As with Autotools-based programs, however, it's well worth checking out the **README** and **INSTALL** files (if they exist) before you do anything.

Extract the program's source code and **cd** into the resulting directory. Then enter the following commands:

```
mkdir build
cd build
cmake .. && make
```

The **&&** here is important, and you might not have come across it before if you don't spend much time at the command line. Basically, it means: only run the following command if the previous command was successful. So only try to compile the code if the configuration step (**cmake ..**) went without any problems. (You'll often see shell scripts where multiple commands are strung together with **&&** symbols, to make sure that everything runs in order and correctly.

After the software has been compiled, you'll need to run the **make install** step as root, as described in the main text (using **su root -c** in Debian and **sudo** in Ubuntu). The files will be copied into your filesystem, and you can run the program using its name.

because Alpine is highly portable and runs on many different operating systems, so if you're compiling it for a fairly obscure Unix flavour you may need to disable some features.

Now, there may be nothing that particularly takes your fancy, so you can run the configure script on its own like so:

```
./configure
```

But **configure** also does something else that's important: it makes sure that your system has everything needed to compile the program. For instance, if you don't have GCC installed, you'll see an error message saying that you don't have a compiler. On Debian and Ubuntu-based systems, a quick way to get the basic packages required for compiling software is to install the **build-essential** package. On Debian (you'll be prompted for your root password):

```
su root -c "apt-get install build-essential"
```

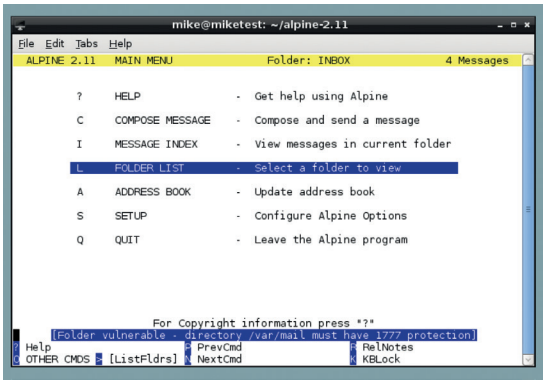
And on Ubuntu (you'll be prompted for your normal user password):

```
sudo apt-get install build-essential
```

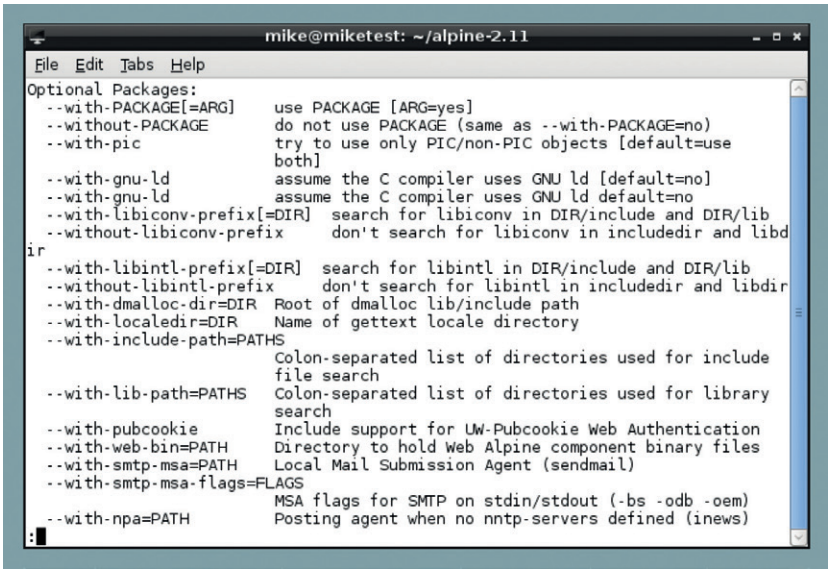
Now run **./configure** again and see if any other error messages come up. This is where it can start to get a bit messy, thanks to the complicated world of dependencies – that is, external libraries that the program depends on. For instance, on Debian we got an error of "Terminfo/termcap not found". That's not especially useful, as it doesn't tell us which package we need. But 20 seconds of Google searching for that error message provided a solution: we need to install **libncurses5-dev**.

Finding dependencies

A similar error popped up for PAM, which we resolved by installing **libpam-dev**. Ultimately there's no magic way to solve dependency issues, but you can usually get by with the **README**/**INSTALL** files, Google and **apt-cache search** to find packages with names relating to the error messages (you usually need ones that end in **-dev** to compile programs from the source). If you get completely stuck, try asking on the program's forum or mailing list, or even try contacting the developer directly. You may even politely suggest that he/she includes a list of dependencies in the



And here it is: our freshly baked, self-compiled Alpine mail client. It's not much to look at, but take it from us, it's a very fine mailer indeed.



Using the **configure** script you can customise the installation locations and enable experimental or advanced features.

README file in subsequent versions of the program... Once the **configure** script has run without any hitches, enter the most important command of all:

```
make
```

This does the compilation job, and depending on the size of the program, it could take minutes (a small command line tool) to hours or days (LibreOffice). So grab a cuppa and check back in periodically. Once the compilation is complete, you'll need to copy the files into your filesystem – this requires root (admin) privileges. So on Debian:

```
su root -c "make install"
```

And on Ubuntu:

```
sudo make install
```

And that's it! Start the program by typing its name on the command line – eg **alpine**. And enjoy the warm fuzzy feeling of running a program that was compiled on your own machine, with your own patches and options, because you're no longer a slave to the distro vendors. You can now grab and install programs before someone packages them up, and you'll find it much easier to try the applications that we feature in our FOSSpicks section. Happy times indeed.

If you're a developer, you can use GNU Autotools to provide the same configure script and Makefile setup that many other programs use. This is better than rolling your own build scripts, as distro packagers prefer using established and well-known systems. An excellent – albeit extremely lengthy – tutorial can be found at **http://autotoolset.sf.net/tutorial.html**. You can ignore much of it (especially the sections on Emacs if you don't use that editor), so skip down to the part that's headlined "The GNU build system". This is another name for Autotools, and the guide there will show you how to put the right files in place and set up their contents correctly so that users can simply run **./configure**, **make** and **make install** as normal.

Mike Saunders has been compiling stuff for more than 15 years, and once compiled a compiler for the ultimate recursive experience.