COMPILING LINUX SOFTWARE FROM SOURCE CODE

A computer program is a list of instructions given to a computer to make it perform a specific task or series of tasks. Computers do not understand English (we all wish that they did!), so programmers must communicate these instructions to the computer in a language the computer understands. Computers, however, can only operate on numbers, which makes a computer's language very difficult for humans to understand.

The solution to this problem is to create an intermediate language that both humans and computers can understand. These are called programming languages. Programmers create a list of instructions for the computer in a programming language such as C, Pascal, or Fortran. This list of instructions is known as *source code*. It is textual in nature, and readable to humans (who speak the language). Programmers do all their work in this source code, changing the instructions to fix bugs, add features, or alter the appearance of a program.

When the programmer believes he has perfected the instructions for his program, he uses a special program called a *compiler* to translate his human readable text instructions into computer readable numbers that correspond to the same instructions. The resulting file is usable by computers but incomprehensible to humans. This is called *object code*. The resulting executable file is often called *binary*, after the number system used by the computer. This translation from source code into binary object code is a one-way process. It is impossible to translate a binary executable back into the original source code.

The binary executable is what you need if you want to run and use a program. This is commonly what you will receive when you purchase shrink-wrapped software from a retail store. The source code is what you need if you want to understand how a program works internally, or if you want to change, add to, or improve a program. If you have the source code and an appropriate compiler, you can produce the binary executable, but the reverse is not true.

If you've downloaded a source code archive, you'll have to compile it before you can install it. In the top level directory of your extracted source, there is usually a README or INSTALL file that gives you instructions on how to compile and install the package. Some instructions are better than others, and rarely will they tell you everything you need to know. Hopefully this guide will help fill in some of the blanks.

A Quick Example

A typical incantation to install a source package looks something like this.

```
[root]# tar -xzvf ${name}.tar.gz

Unpacking ...

[root]# cd ${name}

[root]# more README

[root]# more INSTALL

[root]# ./configure

[root]# make

[root]# make install
```

Unpacking the Archive

```
[user]$ tar -xzvf ${filename}.tar.gz

[user]$ tar -xjvf ${filename}.tar.bz2

[user]$ tar -xvf ${filename}.tar

[user]$ unzip ${filename}.zip

[user]$ unzip ${filename}.exe
```

Depending on the type of archive you have received, the command for unpacking it may differ slightly. Shown above are some popular examples. `tar` is a Unix utility (Tape ARchiver) for working with bundled file archives. Typical command switches are `x` to extract files, `v` for verbose mode so you can tell what's going on, `f` indicating there will be a filename to follow. If the file name ends in `.tar` that is all you need, but usually the tar file has been compressed with another utility. Files ending in `.gz` were compressed with `gzip`. For those, add the `z` flag to the list of parameters. Files ending with `.bz2` were compressed with the `bzip2` utility. For those, add the `j` flag, or failing that, `--bzip2`.

Occasionally you will run into an archive delivered in a `.zip` file. Use the `unzip` utility to unpack these. (Note that you may have to install the unzip package first.)

Some files you download may be built into self-extracting zip archives that end in `.exe`extension. Usually `unzip` can unpack those files as well, but be warned that anything packaged this way is almost certainly meant for a Windows machine and not your Linux box. Programs packaged this way probably will not run on Linux. But you might have an archive that contains only fonts or documentation, and those should be okay.

Configuring

The typical steps to configure and install software are these.

```
[root]# ./configure --help

[root]# ./configure

[root]# ./configure --prefix=/home/vince
```

Why do you type "dot slash configure" instead of just "configure"? Because the configure script is in the current directory, represented by the dot. By default, Linux does not search the current directory for executables; you must explicitly tell it where to look. (This is a security feature.) If you don't type the dot slash, you'll get an error like

<samp>bash: configure: command not found</samp>, and may spend hours pulling your hair out trying to figure out why.

The primary job of the configure script is to detect information about your system and “configure” the source code to work with it. Usually it will do a fine job at this. The secondary job of the configure script is to allow you, the system administrator, to customize the software a bit. Running `./configure --help` should give you a list of command line arguments you can pass to the configure script. Usually these extra arguments are for enabling or disabling optional features of the software, and it is often safe to ignore them and just type `./configure` to take the default configuration.

There is one common argument to `configure` that you should be aware of. The `--prefix`argument defines where you want the software installed. In most source packages this will default to `/usr/local/` and that is usually what you want. But sometimes you may not have root access to the system, and you would like to install the software into your home directory. You can do this with the last command in the example, `./configure --prefix=/home/${vince}` (where ${vince} is your user name).

Compiling, Installing and Uninstalling

```
[root]# make

[root]# make install

[root]# make uninstall
```

The next step is to invoke the GNU Make utility to read the Makefile and compile the program for you. Unlike the Linux shell, `make` *does* look in the current directory for its Makefile, so you needn’t specify anything else. You should be aware that compiling software can take a long time. Compiling a simple program may take only a minute or two, but if you are planning to compile all of KDE from source, you may have to wait hours or even *days* depending on the speed of your computer. Also, it is not at all

unusual to see hundreds of compiler warnings scroll by while software is compiling. If you are lucky, the software will compile anyway.

Assuming the compile phase completes without error, the next step is to actually install the software using `make install`. This invokes the `make` utility again, this time using it to copy the newly compiled files where they need to be in order to run your program. (See **Where did the files go?**) With some programs, you can remove the installed files using `make uninstall`as well, but this is not universal.

Most errors you will bump into while compiling have to do with missing libraries that the software depends on. Every case is unique, but watch for “not found” or “unable to locate” phrases. Typically you just need to install the “development” versions of the libraries it needs. These are usually available from your operating system vendor packages. Search for packages with names ending in “-devel”.

Source : http://www.control-escape.com/linux/lx-swinstall-tar.html