The Answer Girl
INSTALLATION WITHOUT TEARS

There is one main drawback to self-compiled software: whether or not it can be neatly uninstalled later depends on your own discipline. As Patricia Jung explains, there is a remedy at hand

Anyone who installs software via the rpm or deb packages pre-compiled for their respective distribution does so in the safe knowledge that, if necessary, they can also be removed in the same manner. Even is the process does leave behind dead files on the system, this doesn't detract from the feeling of being blameless: if this happens then it was the package compiler who did the sloppy work.

Even distributions well-equipped in terms of software have to pass at some point: not every useful program comes shake-and-bake for all distributions and distribution versions. When the words "self-compiling" appear, the moment of truth will arrive by the time you get to *make install* if not before. If you don't keep track precisely, you'll may well find the installed binary and possibly the corresponding manpage, but was that really all that this inconspicuous command – executed with *root* rights – has tucked away in the depths of your filesystem?

Work ban for make

There is a manpage for *make*, however, and this explains that the *-n* option lists all commands that come in to play during the handling of the **Makefile**, although it doesn't execute them. That's no problem, so long as this list is as clear and simple as it is in the **mirror** package:

```
pjung<chekov:~/software/mirror$ make -n install
[...]
install -m 755 -g gnu mirror.pl
/usr/local/sbin/mirror
[...]
```

Makefile: A file containing the instructions that "make" should execute. If the GNU version of make is not told, with the option -f, which file this is meant to be, it searches methodically for a file named GNUmakefile, makefile or Makefile in the working directory.

mirror: A tool which "reflects" directory hierarchies from one computer on to another, so that there are identical copies.

The Answer Girl

The fact that the world of everyday computing, even under Linux, is often good for surprises, is a bit of a truism: Time and again things don't work, or at least not as they're supposed to. The Answer-Girl in Linux Magazine shows how to deal elegantly with such little problems.

Only the *install* command is invoked here, which copies the file "mirror.pl" into the target directory */usr/local/sbin* and at the same time renames it as *mirror*. There, using the mode option *-m 755* it is given the rights *rwxr-xr-x* and is assigned to the gnu group (which must already exist) using *-g*.

If the commands to be executed by *make install* are copied with the re-routing operator > into a file to be archived (*make -n install > filename*), the installation process can be reconstructed if you later go on an uninstallation binge and delete */usr/local/sbin/mirror* and its consorts from the system by hand with the *rm* command .

As soon as *make install* has to execute long-winded, highly-complex shell scripts, which in turn rely on the scripts which come with the source package, the will to discipline will certainly no longer suffice. A different approach is needed.

rwxr-xr-x The *ls* option *-l* shows which users can do what with a file or a directory – *r*, "read", *w*, "write", execute (in the case of directories: change to, *x*, "execute") – or not (–). The first trio of letters, rwx, says that the file owner can do everything. The group to which the file is assigned, on the other hand, only has read and execution rights – it cannot alter (write) the file on the grounds of the "–" in the middle trio. All other users, too, only have the rights to read, execute or change to. A different notation for rwxr-xr-x is 755. This is produced by coding r with 4, w with 2, x with 1 and – with 0 and adding the values of a trio: 4+2+1=7, 4+0+1=5.

To each his own directory

The simplest solution would be to accommodate each self-compiled software package in a directory of its own. The "*F*ilesystem *H*ierarchy *S*tandard" (*http://www.pathname.com/fhs*) suggests */opt/packagename* for "add-on application software packages". However, another sensible alternative is that of a package directory under */usr/local*, even if there is no provision for this in the FHS.

In this package directory a typical Unix directory hierarchy with *bin* directory for user programs, *sbin* for the binaries reserved for the system administrator "root", *man* for manpages, *lib* for libraries, *include* for any **header files** etc. should then be made, provided the software package concerned comes with suitable files.

So how do you persuade *make install* to copy into such a directory? Apparently by modifying the file *makefile* or *Makefile*. Once a back-up copy has been made, it's time to search for the *Target install*. The Makefile does in fact specify the arguments with which *make* can be invoked: only words that stand at the start of a line therein and end in a colon are permitted. The *makefile* from the *mirror* package is very clear:

```
install:
[...]
        install -m $(EXMODE) -g $(GRP) ➥
mirror.pl $(BINDIR)/mirror
[...]
```

Here we find, indented by precisely one tab character, the commands to be executed by *make install* – although concrete values are replaced by variables. A comparison shows that *$(BINDIR)* is obviously inserting the content of the variable *BINDIR* at this point, and a few lines higher up *BINDIR* is assigned the value */usr/local/sbin*:

```
# directory to install public executables
BINDIR = /usr/local/sbin
```

If we want to change the "directory to install public executable files" into */opt/mirror/sbin*, all it takes is one correction in the *makefile*:

```
BINDIR = /opt/mirror/sbin
```

The challenge in this method consists only of finding all variables relating to the installation step. This can turn into production-line work in which many control processes in terms of *make -n install* become necessary. It often happens with such hand-written makefiles, too, that the authors have not done tidy work and have "unintentionally" included static path specifications too. The only remedy for this is a methodical search for the stubborn directory specification.

There's another problem too; if *make install* complains, with an error message such as:

```
install: cannot create regular file ➥
`/opt/mirror/sbin': No such file or directory
```

then the target directory */opt/mirror/sbin* is missing and must be made by hand. If the parent directory *mirror* is missing as well as its subdirectory, *sbin*, then you can save yourself a *mkdir* command if you use:

```
mkdir -p /opt/mirror/sbin
```

to make all the necessary *p*arent directories at once.

Configure taken at its word

With software projects above a certain size it becomes too tedious for most authors to wait for Makefile by hand. In these cases they depend on automatic Makefile production mechanisms. The drawback here is that the Makefile gets very complex – one reason why *make -n install* becomes difficult to decode.

This is offset by the advantage that the Makefile creation tool is very easy to influence. If the *configure* script that comes with a software package correctly bears its name, it should also be possible to specify the place to which the files already created should be copied.

In fact *configure* speaks to the common help option *––help* – Listing 1 shows extracts from the Help for the *configure* script of the mail program *sylpheed*. Equipped with this information, with:

```
./configure --cache-file=/tmp/sylpheed.tests ➥
--prefix=/opt/sylpheed --enable-ssl
```

we ensure that SSL support is included and all files are stored on installation with *make install* under */opt/sylpheed*. *––cache-file* allows a file to be specified after the equals sign, in which the test results found by *configure* are to be saved. Normally the *config.cache* is in the current directory but here

/usr/local This directory, which ideally sits on its own partition, is intended to keep locally installed software on hand. "Local" here means both "not part of the distribution", and also (in networks) "actually installed on the hard drive of this computer and intended only for this computer" (unlike centrally stored resources which may be made available via the "Network File System" or NFS of the individual workstations).

Header files If you want to compile software which uses the functionality from libraries dynamically, the interfaces of this library – the API (Application Programmer Interface) –, must be at hand when the time comes to compile. In C and C++ programs these are found in so-called header files with the ending .h.

we have instead selected */tmp/sylpheed.tests*.

Anyone who now goes off and makes changes in the *Makefile* created, by the way, is doing this with a safety net in place: should any manual corrections turn out to be wrong, all you need do, instead of the whole *configure* rigmarole, is say *./config.status*. This executable file saves precisely what needs to be done to reach the same conclusion as the last *configure* run.

A *make* compiles the software, while *make install* makes an orderly Unix file tree under the prefix directory and copies in the files needed to use the software. In the case of Sylpheed, a *bin* subdirectory is created under */opt/sylpheed* containing the executable program, plus a *share* directory, which contains the online manual in HTML format.

Since */opt/sylpheed/bin* does not lie in the search path which can be displayed with *echo $PATH*, a simple *sylpheed&* will not work (or starts a program of the same name, which may already be in one of the directories listed in the *PATH*). Only when the full path to the binary is specified...

```
/opt/sylpheed/bin/sylpheed&
```

... does the new program start.

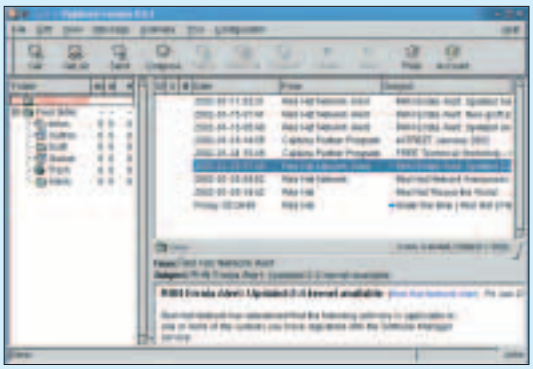

Figure 1: Sylpheed running

Listing 1: Help for a configure script

```
pjung<chekov:~/software/sylpheed-0.6.5$ ./configure --help | less
Usage: configure [options] [host]
Options: [defaults in brackets after descriptions]
Configuration:
--cache-file=FILE cache test results in FILE
--help  print this message
[...]
Directory and file names:
--prefix=PREFIX  install architecture-independent files in PREFIX
                 [/usr/local]
[...]
Features and packages:
[...]
--enable-ssl     Enable SSL support using OpenSSL [default=no]
[...]
```

Forging paths

With an

```
export PATH=$PATH:/opt/sylpheed/bin
```

the *PATH* variable can be extended quickly and simply into a shell. Use *$PATH* to fetch the previous content of *PATH* and attach the new search directory following a colon. The result is in turn assigned *PATH* as new value. *export* ensures that the bash passes on the variable to "child processes", such as to a *console*, which is started via *console &* from the current shell.

If you have already found a *sylpheed* binary in the old path, this resetting will be precious little help, since the shell always takes the file it finds first. If on the other hand, when resetting the path, one places */opt/sylpheed/bin* before all previous search directories:

```
export PATH=/opt/sylpheed/bin:$PATH
```

... the tables are turned, and the binary previously found without specifying the directory is now the one that must be called up explicitly.

But who wants to have to change the path every time he/she needs a program from a "non-standard" directory? If all users of the computer are to get something out of the path extension, it would be advisable to enter the change from *root* in the system-wide configuration file */etc/profile*. (Some distributions, such as SuSE, also read in files specially designed for local changes, such as */etc/profile.local*.)

If the path extension is only intended to apply for your own user account (which makes sense for things like software which one has installed in one's own home directory – such as under *~/bin* –), then the personal start files of the shell (for the *bash~/.bashrc* and *~/.bash_profile*) are candidates for correction.

No new paths

Anyone who does a lot of self-compiling will soon get fed up with the constant path corrections – especially since a multi-line *PATH* monster is no longer easy to grasp. What a good thing there are some alternatives. *Links* in a Linux/Unix filesystem ensure that a file can be addressed by several names. Instead of doing something like copying the *sylpheed* binary from */opt/sylpheed/bin* into the directory */usr/local/bin* and thus having two copies of it, a "symbolic link" is all it takes

```
ln -s /opt/sylpheed/bin/sylpheed ➥
/usr/local/bin/sylpheed
```

to make the binary accessible using either specification.

For fairly small programs like Sylpheed this is enough – but woe betide you if you are dealing with larger software packages, which create several

binaries at the same time has lots of manpages and, in the worst case, also come with their own libraries. Nobody wants to set so many links by hand and you certainly won't want to remove them again if */opt/sylpheed* falls victim to the uninstaller *rm -rf /opt/sylpheed*. Although the mini-program symlinks can ferret out and delete any broken links that point into oblivion, this useful tool is pre-installed on few systems.

Now we can set about looking for a tool that will take over the whole task of link management. Debian users are lucky, since this distribution contains just such a mini-program in the form of *stow*.

```
apt-get install stow
```

automatically downloads the corresponding package (assuming you've got Net access) and installs it immediately, provided *root invokes the command*.

But users of other distributions need not despair. Any Debian software can be downloaded from a Debian **mirror** not only as pre-compiled deb package, but also as an original tarball from the author of the software. The Debian download pages on the Web (Figure 2) offer three links in the lower part under the point *Source Code*: the package specification as a *dsc*-formatted ASCII file, the original *tar.gz* source archive and the source code of the changes which the Debian package builder has integrated into the binary package, as a *diff* output packed with *gzip*.

The tar file is unpacked in the usual way with *tar -xzvf archivefile* (the option *-z* unpacks the *gzip* compression, *-x* extracts the content, *-v* provides a bit more talkativeness in *tar* ("*v*erbose"), and *-f archivefile* states which *f*ile is to be unravelled). With

```
./configure --prefix=/opt/stow
```

in the directory *stow-1.3.2.orig* we also configure *stow* with a target directory of its own */opt/stow*. For once we can do without *make*, since on this occasion there is nothing to compile. *make install* then provides a file structure as in Figure 3.

Figure 2: Debian always comes with the original source code

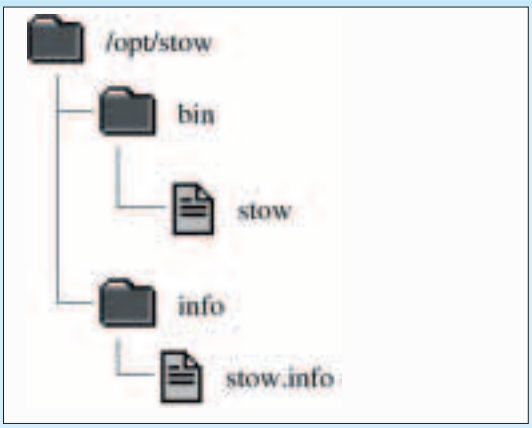

Figure 3: With the prefix /opt/stow, stow gets its own directory hierarchy

Hidden treasures

So now *stow* may be installed, but a bit of documentation wouldn't go amiss. There is in fact an info file in */opt/stow/info*, but anyone not already familiar with this information system will not have much luck with

```
info -f /opt/stow/info/stow.info
```

(see Figure 4). An HTML file, which can be viewed in the browser and printed out neatly formatted, is high on the wish list.

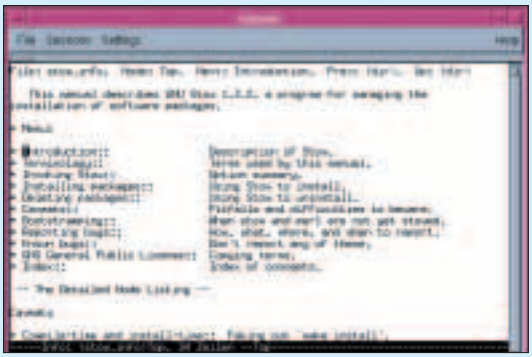

Figure 4: Info with info

The Makefile in *stow-1.3.2.orig* does in fact contain a few useful secrets: even if we aren't *stow* developers, we need not be scared off by the comment

```
# The rules for manual.html and manual.texi ➥
are only used by
# the developer
```

and make a note of the promising Make-Target *manual.html*, which is defined thereafter:

```
manual.html: manual.texi
        -rm -f $@
        texi2html -expandinfo -menu ➥
-monolithic -verbose $<
```

Mirror: A server which "reflects" the data file of another one, thus stores it as a copy which is as up to date as possible. The tool "mirror" is readily used for this purpose.

The first line says that *make* is dependent in these rules on making sure that the rules of the (later defined) target *manual.texi* have been executed, before its own (indented with Tab) commands come into play. The first real action taken by the *manual.html* target consists of deleting any file of the same name which may exist (*$<* stands for the target itself, thus that which is on the line which is not indented before the colon) with *rm -f* ("*f*orce"). Should there be an error message now (perhaps because no such file as *manual.html* exists), *make* should say nothing – hence the minus before the *rm* command.

What really interests us is the second rule: the *texi2html* program is, says its manpage, a "Texinfo-to-HTML-converter". Since *$* in the make-syntax stands for whatever comes after the target name and its colon, it soon becomes clear: this target produces an HTML file from a file (created by the *manual.texi* target) named *manual.texi*.

We can see, in turn, from the *texi2html* manpage, that the program – provided it is invoked using the *-monolithic* option – creates from a Texinfo-file named *foo* a single file *foo.html* (instead of swapping footnotes and tables of contents into additional, individual files).

The command *make manual.html* in the *stow* source directory thus looks precisely as if it is making our wish for a *stow* manual in HTML format come true. There is in fact then a usable *manual.html* in the current directory (Figure 5).

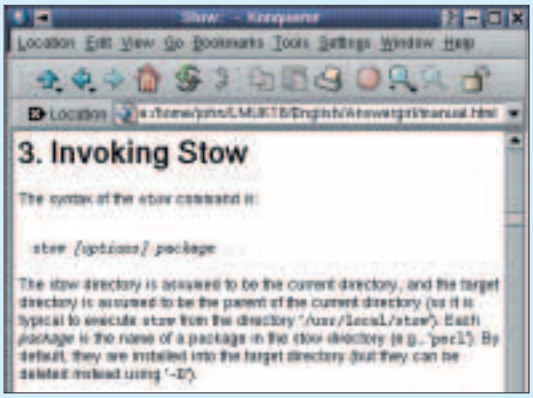

Figure 5: Rewarded by a "make manual.html"

The way it should be

To use *stow* successfully, a few concepts need clarifying. When the documentation talks about the "stow directory", we are dealing with the directory in which the sub-directories with the file hierarchies of the individual, compiled software packages are found. In other words, the preparatory work for using *stow* consists of specifying in each case the prefix *Stow-directory/packagename* in the *configure* run for a software package. In our plan, the stow directory thus bears the simple name */opt*.

The next piece of information *stow* needs to know is the directory hierarchy into which it should link the contents of *stow directory/packagename*. A highly suitable target directory is */usr/local*, especially if */usr/local/bin* is already contained in the *PATH*.

The target directory and stow directory can be specified with the options *-t* and *-d*, and the latter option can be left out if we are dealing with *cd /opt* in the stow directory.

Now we still have to link */opt/stow/bin/stow* itself in orderly fashion to */usr/local/bin/stow*, before we can call it up without specifying the path. An

```
/opt #./stow/bin/stow -v -v -v -n -t / ➥
usr/local stow
Stowing package stow...
Stowing contents of stow
Stowing directory stow/bin
Stowing contents of stow/bin
LINK /usr/local/bin/stow to ➥
../../../../opt/stow/bin/stow
Stowing directory stow/info
LINK /usr/local/info to ../../../opt/stow/info
```

gives us control over what would happen to the content of the directory *stow* if we wanted to link it to */usr/local*: The option *-n* ensures in the meantime that *n*othing happens, while each *-v* ("*v*erbose") makes the program a bit more talkative; but this only goes up to chatter level 3.

If directories which can be found under *./stow* do not yet exist in */usr/local* (*/usr/local/info* for example), the program *./stow/bin/stow* certainly does not make them. However it does make things easy for itself: */usr/local/info* points to */opt/stow/info*. If the respective directory already exists (which it does in the case of the example of */usr/local/bin*), stow sets a link therein to the corresponding file (*/usr/local/bin/stow* points to */opt/stow/bin/stow*). For the source file stow specifies relative paths starting from the target directory.

That looks sensible, so we do a good job with *./stow/bin/stow -v -t /usr/local stow* and take a look at the result:

```
/opt # ls -Al /usr/local
total 3
drwxr-xr-x 2 root root  55 Nov 26 20:02 bin
drwxr-xr-x 2 root root 150 Nov 26 18:28 ftp
lrwxrwxrwx 1 root root  22 Nov 26 20:02 ➥
info -> ../../../opt/stow/info
drwxr-xr-x 2 root root  57 Nov 26 18:28 man
```

The trouble with bugs

Before we get euphoric and start stowing other software, we'd better first check if the promised *De*-installation with the option *-D* really is that simple. If */usr/local/bin* lies in the search path, we can now call up *stow* without specifying the directory:

```
/opt # stow -v -v -v -n -D -t /usr/local stow
Unstowing in /usr/local
Unstowing in /usr/local/bin
Unstowing in /usr/local/ftp
Unstowing in /usr/local/ftp/bin
Unstowing in /usr/local/ftp/dev
Unstowing in /usr/local/ftp/etc
Unstowing in /usr/local/ftp/lib
Unstowing in /usr/local/ftp/usr
Unstowing in /usr/local/ftp/usr/bin
Unstowing in /usr/local/ftp/msgs
Unstowing in /usr/local/man
```

That looks funny – why is */usr/local/info* never mentioned? Why is there nothing saying that */usr/local/bin/stow* is to be deleted? And what has stow lost in subdirectories such as *man* and *ftp*, into which it has linked nothing at all?

The brave will now back up the complete */usr/local* hierarchy and let *stow -D* run again without the *-n* option. But this is no help, either:

```
/opt # ls -al /usr/local/bin
total 16
lrwxrwxrwx 1 root root 29 Nov 26 20:02 ➥
stow -> ../../../../opt/stow/bin/stow
```

The links are still there.

No matter how much we try, here and there, at some point we have to swallow the bitter pill and admit to ourselves: stow is faulty and inadequately tested. It will only really function when the stow directory containing the package subdirectories is itself a subdirectory of the target directory.

It's a good job we know what has been linked:

```
/opt # rm /usr/local/info
/opt # rm /usr/local/bin/stow
```

So we make a new stow directory *stow* under */usr/local* and pack our *stow* package directory into that:

```
/opt # mkdir /usr/local/stow
/opt # mv stow /usr/local/stow/
```

Stowing

The advantage of the new stow directory is that we no longer have to specify the target directory */usr/local* at the same time:

```
/usr/local/stow # ./stow/bin/stow -v stow
Stowing package stow...
LINK /usr/local/bin/stow to ➥
../stow/stow/bin/stow
LINK /usr/local/info to stow/stow/info
```

links correctly, and the deinstallation also looks reasonable:

```
/usr/local/stow # stow -v -D stow
UNLINK /usr/local/bin/stow
UNLINK /usr/local/info
UNLINK /usr/local/bin/stow
RMDIR /usr/local/bin
```

Links are removed and directories which are now empty such as */usr/local/bin* are deleted. After re-linking stow, */usr/local/bin* points, as a newly-made link, to */usr/local/stow/stow/bin*.

To now install *sylpheed* neatly, too, we must however reconfigure and compile the mail program with new prefix */usr/local/stow/sylpheed*, before *stow* can pursue its linking work after a *make install*:

```
/usr/local/stow # stow -v sylpheed
Stowing package sylpheed...
UNLINK /usr/local/bin
MKDIR /usr/local/bin
LINK /usr/local/bin/stow to ➥
../stow/stow/bin/stow
LINK /usr/local/bin/sylpheed to ➥
../stow/sylpheed/bin/sylpheed
LINK /usr/local/share to stow/sylpheed/share
```

... and this will give you a nice surprise:

```
/usr/local/stow # ls -al /usr/local/bin
total 0
lrwxrwxrwx1 root root 21 Nov 26 20:13 stow ➥
-> ../stow/stow/bin/stow
lrwxrwxrwx1 root root 29 Nov 26 20:13 ➥
sylpheed -> ../stow/sylpheed/bin/sylpheed
```

Suddenly */usr/local/bin* is no longer a symlink to the *bin* directory of the stow package, but an ordinary directory, in which two new symlinks can be found: After the old link to */usr/local/stow/stow/bin* was broken, the stow program was simply linked independently. If we should now yearn to get rid of *sylpheed* again, everything rolls itself neatly back again:

```
/usr/local/stow # stow -v -D sylpheed
UNLINK /usr/local/bin/sylpheed
UNLINK /usr/local/share
UNLINK /usr/local/bin/stow
RMDIR /usr/local/bin
LINK /usr/local/bin to stow/stow/bin
/usr/local/stow # ls -al /usr/local/bin
lrwxrwxrwx 1 root root 13 Nov 26 20:14
/usr/local/bin -> stow/stow/bin
```

An *rm -rf /usr/local/stow/sylpheed* then ensures that (apart from personal mail and configuration files) there really are no remains left behind on the system.

. Short notation for the shell for the directory, in which one is currently at. Two dots (..) on the other hand designates the "parent directory" lying exactly above the working directory.

ls -A The option -A makes sure that ls also lists "hidden files", whose names begin with a dot. Contrary to -a, the user does not also see the current (.) and the parent (..) directory listed at the same time.

Info

mirror
http://sunsite.org.uk/packages/mirror
sylpheed
http://sylpheed.good-day.net
symlinks:
http://packages.debian.org/unstable/utils/symlinks.html
stow
http://packages.debian.org/stable/admin/stow.html