Handling of complex tasks with make

MAKE AND BAKE

Developers, LaTeX users, and system administrators can all harness the power of make.

BY HEIKE JURZIK

Anyone who has compiled a program from the source code will be familiar with the make tool, the central part of the *configure*, *make*, *make install* three-card trick for building and installing software on Linux. This is not the only situation in which make provides useful service. Make can also assist with other tasks, for example, in larger projects, LaTeX users can use make to automatically recompile and create a Postscript or PDF document if one or multiple source files have changed.

Make also helps sysadmins, and it can automatically launch a backup script if you have modified one or more files. A control file, makefile, describes the chores you want make to handle. After setting up the makefile, you can simply call *make* – and let it do all the work.

The make program is available for a variety of platforms. In this article, we will be looking at GNU make, which is used in the Linux world. I'll investigate the makefile layout and introduce the command's most interesting options.

A Question of Control

The makefile is the control point for working with make. Each makefile holds a list of instructions. Make expects a *GNUmakefile*, *makefile*, or *Makefile*, and it looks for the control file in this order in the current directory. The makefile contains rules that follow this syntax:

```
Target: Dependency (ies)
      command
      command
      ...
```

The target refers to the expected result of the commands that follow. The colon is followed by one or multiple files required to build the target, that is, on which the target depends. This list is followed by the actions that make will perform, provided the dependencies are fulfilled.

Keep to the following syntax rules when creating makefiles:

- Comments are prefixed by a pound sign (#).
- Dependencies are blank separated and follow the colon that follows the target.
- Each command occupies a separate line. Commands are typically tab indented; if you use blanks, make might refuse to work, and you will see a message such as *makefile:4: *** missing separator. Stop.*
- If you prefer to list the commands in a single line, use semicolons to separate them. Tab indenting is mandatory.
- You can wrap long lines by inserting a backslash (\) at the end of the line. The backslash has to be the last character in the line.

A Great Team – Make and LaTeX

To demonstrate make in a practical situation, let's look at a makefile that helps users create books with LaTeX. The book is stored in multiple documents; in other words, there is a Tex file for each chapter. LaTeX will compile these components to create a file called *book.tex*.

First create the **DVI** file. We want make to call the *latex* command whenever the source files change after this.

```
book.dvi: chap01.tex ➥
chap02.tex chap03.tex \
      chap04.tex chap05.tex ➥
      chap06.tex \
      chap07.tex book.tex
      latex book
```

We will be using the Dvips program with a variety of parameters to automatically

create a Postscript file. The tool has nothing to do, unless the DVI file for the first target has changed:

```
book.ps: book.dvi
        dvips -q -o book.ps book
```

If you like, you can add another target to create a PDF document from the Postscript file:

```
book.pdf: book.ps
        ps2pdf book.ps book.pdf
```

A good backup is definitely worthwhile after all that work. It makes sense to bundle the Tex files, the Postscript file, and PDF into a compressed Bzip2 tar archive, and then run SCP to copy it to a remote machine – better be to safe than sorry:

```
backup: chap01.tex chap02.tex ➥
chap03.tex \
      chap04.tex chap05.tex ➥
chap06.tex \
      chap07.tex book.tex b➥
uch.ps \
           book.pdf
     tar cvfj backup.tar.bz2 ➥
*.tex book.ps book.pdf
       scp backup.tar.bz2 huhn@➥
asteroid.huhnix.org:
```

Make at Work

There are various options for using makefiles like this. If you run *make* without passing in any other parameters, make will simply process the first target, that is, a DVI file if you have modified one or more Tex files.

As an alternative, you can pass a different target in to make when you call the tool, for example *make backup*, to create a safe copy, or *make buch.pdf* to create the PDF document.

Just as a reminder: make creates the PDF from the Postscript file by calling the *ps2pdf* command; the Postscript file is created from the DVI file, which is the makefile's first target. If anything has changed on the way to the Postscript file, make will process all the targets from the top down (Figure 1). If there is nothing to do, make will let you know:

```
make: `buch.pdf' is up to date.
```

GLOSSARY

DVI: Device Independent output format is an output file developed for the TeX text system to store the results of the latex file.tex command. This format is suitable for screen viewing, or printing.

Figure 1: Looking for changes with make.

A Question of Options

Make has a number of other command line parameters. If the control file uses a name that make does not associate with a makefile, you need to pass in the filename, along with instructions for processing it, by setting the *-f* flag:

```
make -f controlfile
```

It is also possible to specifically name the working directory. If you are currently at a different location on the filesystem, you can use the *-C* parameter to tell make where to go to work:

```
$ make -C ~/book
make: Entering directory ➥
`/home/huhn/book'
latex book
...
make: Leaving directory ➥
`/home/huhn/book'
```

If the source files haven't changed, you can "force" a build with the *-B* option:

```
$ make -B book.pdf
latex book
...
Transcript written on book.log.
dvips -q -o book.ps book
ps2pdf book.ps book.pdf
```

And if you are interested in finding out what is going on under the hood, you can set the *-d* (for debug) option for verbose output (Figure 2). To prevent the output from scrolling off the screen, you might like to pipe it to a pager such as less or more:

```
make -d | less
```

Versatile

The LaTeX example uses a couple of Tex files, which keeps the makefile fairly readable. When the number of source files starts to grow, it is easy to lose track. To prevent this, make gives you the option of defining variables.

To assign a variable called *TEXFILES* to the LaTeX files, you define the variable at the start of the Makefile:

```
TEXFILES = chap01.tex chap02➥
.tex chap03.tex \
            chap04.tex chap05➥
.tex chap06.tex \
            chap07.tex book.➥
tex
```

Then reference the variable in the makefile, as *$(VARIABLENAME)*, for example:

```
book.dvi: $(TEXFILES)
        latex book
```

Make also has some standard variables. For example, *$@* refers to the current target, and *$?* refers to modified dependencies, that is, to the files that follow the colon. ■