

Department of Information Technology and Electrical Engineering

**VLSI I:  
From Architectures to VLSI Circuits and FPGAs**

227-0116-00L

Sample Solution Exercise 0

---

**Introduction to the Linux Shell**


---

Prof. L. Benini  
F. Gürkaynak  
M. Korb

Last Changed: 2023-09-18 21:25:53 +0200

# 1 Introduction and Notation

In this exercise, you will learn the basics of working effectively with the Linux shell. The content of this exercise is not specific to digital design but highly helpful in working with Linux machines in general.

Throughout this exercise, you will be mostly entering statements into the shell. To open a shell (also called “terminal”, “console”, or “command prompt”) on a GNOME desktop, open the launcher menu by pressing the  key or move the mouse to the upper left corner of the screen, search for the “Terminal” application and start it. There might be multiple “Terminal” applications installed; they only differ in the user interface, but not in the underlying shell.

Shell commands in the exercise sheets are distinctly formatted in a box, where every statement starts with `sh>` (do not type that). For example:

```
sh> echo 'I can make my computer talk.'
```

In explanations we use the same style in-line with text. In this trivial example, `echo` is the command that prints all its arguments, and its single argument (multiple words surrounded by quotes) is the string `'I can make\ my computer talk.'`<sup>1</sup> Paper is finite, so sometimes we have to break a line in a statement where you should *not* begin a new line. In this case, we end the line with the `\` backslash, which is slightly different from the regular code backslash `\`. You should enter the statement *without that backslash or line break*.

**Student Task:** Parts of the exercise where you are required to take action will be explained in a shaded box, just like this paragraph.

**Student Task 1:** Try the statement above once with backslash and newline and once without, i.e.,

```
sh> echo 'I can make my computer talk.'
```

```
sh> echo 'I can make my computer \  
talk.'
```

Make sure that you use *single* quotes ( `'` ) to preserve the backslash literally.<sup>a</sup> What do you observe when entering the statement? How does the output change?

**Sample solution:** The string enclosed in single quotes is printed literally. In the second case, this includes the backslash and the line break, which is undesired most of the time.

<sup>a</sup> The difference between single and double quotes will be explained in Note 6.

You will also be working with files and directories denoted by their *paths* (more on this shortly). We write paths like this: `some_directory/some_file.txt`, and the same rules for breaking overly long lines apply.

## 2 Preparation

In the exercises, you will always start in a preconfigured working directory.

**Student Task 2:** Setup the working directory for this exercise by calling our install script:

```
sh> /home/vlsil/ex0/install.sh
```

This will create the directory `ex0` under your current working directory and populate it. Switch into the newly created directory by entering

<sup>1</sup> `echo` can also be invoked with multiple arguments. If you pass multiple arguments, `echo` prints all arguments with one space between them. Therefore, `echo Hello there!` will print the same as `echo 'Hello there!'` but different from `echo 'Hello there!'`.

```
sh> cd ex0
```

that is, “change directory to **ex0**.”

### 3 Command Line Basics

Within this working directory, we will now look at the basics of creating, manipulating, and deleting directories and files. If you mess up your directory, you can always re-run the install script above to get a fresh one.

You are probably used to navigate and modify directories in a graphical program like Windows Explorer or MacOS Finder. With a few simple commands described in this section, you can do the same tasks on the shell. In Section 5, you will then compose these commands to perform tasks that you cannot do in graphical programs.

#### 3.1 Working with directories

As you might have guessed from the previous tasks, you will always be in a “current working directory”. When starting a new terminal, you will start out in your home directory, from where you can navigate to another directory. The absolute path of your home directory is `/home/$USER`, where `$USER` is your user name. The abbreviation for your home directory is `~` (tilde); you can thus address files and directories under your home directory with `~/dir/file`. Paths under your current working directory can also be abbreviated as relative paths without a leading `/` (slash), e.g., `dir/file` is equivalent to `/home/$USER/dir/file` when you start a new shell.

##### Student Task 3: Enter

```
sh> pwd
```

which is short for “**p**rint **w**orking **d**irectory”. Which directory are you currently in? Does `pwd` output absolute or relative paths?

**Sample solution:** `pwd` prints absolute paths. If you followed the preparation instructions, you should be at `$HOME/ex0`, where `$HOME` is the path to your home directory (can be printed with `echo ~`).

##### Student Task 4: Enter

```
sh> cd examples
```

to change to the `examples` directory and list the files and directories there with

```
sh> ls
```

which is short for “**l**ist”. Do the same for the root directory, i.e., `/`:

```
sh> ls /
```

How do the two directories compare?

**Sample solution:** The `examples` directory contains user-generated files and directories, which have long, human-comprehensible names. The root directory, `/`, is the origin of the local Linux directory tree (see Fig. 1), and its subdirectories have short, standardized names.

A typical directory tree for a Linux system could look like in Figure 1. Note that the ETH lab machines might have a slightly different layout, as your home directory will not be stored on the machine itself, but on the network!

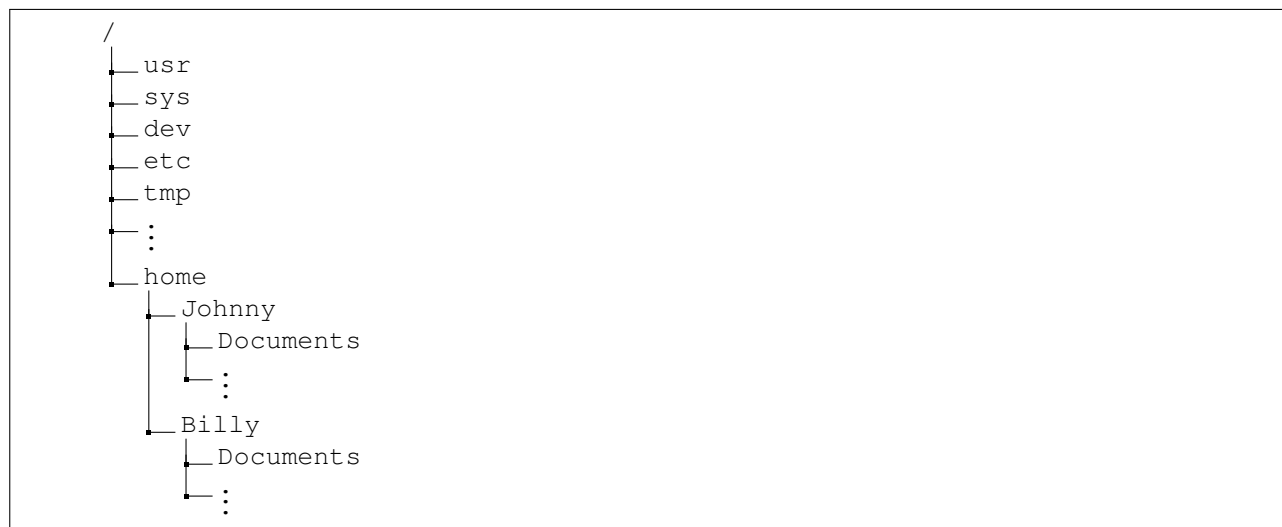


Figure 1: Typical Linux directory tree

#### Student Task 5: Go back to the exercise directory with

```
sh> cd ..
```

and enter

```
sh> tree
```

What does this command do and how is it similar to a command you saw before? How is it different from that command? You may have to use your mouse wheel to scroll the shell window.

**Sample solution:** Like `ls`, `tree` lists contents of a directory, but it does so in a tree-like format that is nicely suited for human comprehension, but not for machines to parse.

Command	Description
<code>pwd</code>	Display the current working directory.
<code>tree \$dir</code>	Get a visualization of the directory tree of the given directory.
<code>ls \$dir</code>	List all files and directories in the given directory.
<code>cd \$dir</code>	Change the current working directory to the given directory.

Table 1: Basic commands for navigating around directories. `$dir` is a path you provide.

**Student Task 6 (Creating directories):** Make sure that you are in the `~/ex0` directory. Think of a creative directory name and create a directory with that name with

```
sh> mkdir $dir
```

where `$dir` is the name you thought of and `mkdir` is short for “**make directory**”. Could you successfully create a directory? Try to create a directory that has spaces in its name; what happens? What changes if you do or do not surround the name containing spaces with quotes (‘ or ’)? What happens if you try to create a directory that already exists?

**Sample solution:** Creating a directory that does not yet exist and whose name contains only alphanumeric characters should work fine. If the name shall contain special characters, you have to enclose it in quotes (see Note 6 for quoting rules). If the directory already exists, you get an error.

The `mkdir` example above shows that arguments to a command are separated by spaces in the shell. To have the shell interpret a string with spaces as one single argument, you have to surround the string with quotes.

**Student Task 7 (Removing directories):** Change to the `examples` subdirectory and remove both the `remove_me` and the `try_to_remove_me` directories with

```
sh> rmdir remove_me try_to_remove_me
```

where `rmdir` is short for “remove directory”. Does that work? If no, why not?

**Sample solution:** `rmdir` only deletes empty directories. `remove_me` is empty, so it gets deleted, whereas `try_to_remove_me` is not empty, so you get an error message.

To avoid nasty surprises, `rmdir` does not delete non-empty directories. To remove a directory *and* its contents, use the `rm` command with the `-r` (“recursive”) option. Be careful, though, because there is no “trashcan” in the shell; that is, there is no guaranteed way to recover removed files!<sup>2</sup>

**Student Task 8 (Removing non-empty directories):** What do you have to enter to remove the non-empty directory `try_to_remove_me`? Verify your answer by trying it. What happens if you use the right command without an option?

**Sample solution:** Use `rm -r` to recursively remove a non-empty directory (as explained above): `rm -r try_to_remove_me`. `rm` without the `-r` option cannot be used to remove directories (not even empty ones).

The `cp` (“copy”) command is used to copy files and directories. It takes two arguments, first the source and then the destination. Similarly to `rm`, you have to use it with the `-r` option, which comes before the arguments, to copy directories.

**Student Task 9 (Copying directories):** Which statement creates a copy of the directory `copy_me` called `new_dir`? Verify your answer by trying it and use `ls` on both directories to ensure that all contents are copied. What happens when you execute the same statement again?

**Sample solution:** Use `cp -r` to recursively copy a directory (as explained above): `cp -r copy_me new_dir`. Be careful: When the directory `new_dir` already exists, `copy_me` is copied *into* `new_dir`. (This is the case when you execute the statement above twice.) Even worse, if the directory `new_dir/copy_me` already exists, it gets overwritten without warning by executing the statement above! (You can try this by executing the statement a third time.)

The `mv` (“move”) command is used to move or rename files and directories. While it takes a source and a destination argument like `cp`, there is no `-r` option for `mv`.

**Student Task 10 (Moving and renaming directories):** Move the directory `move_me` into `new_dir`, check the result with `ls new_dir`, and note the statement you used.

**Sample solution:** `mv move_me new_dir`

In a second step, rename the directory `rename_me` to `new_dir`. Which statement would you use? What is the problem with that statement?

<sup>2</sup> Nonetheless, `rm` is *not* a secure way to delete files!

**Sample solution:** Since `new_dir` already exists, `mv rename_me new_dir` would *move* `rename_me` into `new_dir` rather than renaming it. Thus, you must either rename `new_dir` first or use the `-T` option described below.

**Note 1:** The default behavior of `mv` is closer to “move” than to “rename”. Thus, in the particular case when you want to rename a directory to a name that already exists in the same parent directory, `mv` *moves* the source directory into the target instead of renaming (and overwriting) it. You can force a rename with `mv -T`, though.

**Note 2:** As a rule of thumb, both `cp` and `mv` come with no or little safeguards (just like `rm`). Most importantly, if the target of a `cp` or a `mv` already exists, it is silently overwritten in almost all cases!

Command	Description
<code>mkdir \$dir</code>	Create a new directory with the given name.
<code>rmdir \$dir</code>	Remove a directory. Will not work with non-empty directories.
<code>rm -r \$dir</code>	Remove a directory and its contents. <b>Warning:</b> There is no trashcan on the command line! Files and directories will be deleted irrevocably.
<code>cp -r \$source \$target</code>	Copy a source directory (recursively with all its contents) to a target.
<code>mv \$source \$target</code>	Move a file or directory. Also used to rename files/directories.

Table 2: **Basic commands for creating, modifying, and deleting directories.** `$dir`, `$source`, and `$target` are paths you provide.

`rm`, `cp` and `mv` can be used for both files *and* directories. This is because in Unix, directories are just some special kind of file.

## 3.2 Working with files

Command	Description
<code>cat \$file</code>	Output a file to the terminal.
<code>head \$file</code>	Output the first lines (i.e., default value is ten) to the terminal.
<code>tail \$file</code>	Output the last lines (i.e., default value is ten) to the terminal.
<code>less \$file</code>	Browse a file in a visual viewer.
<code>touch \$file</code>	Create a new file when it doesn’t exist yet. If the file already exists, its access and modification times are updated without changing the file content.

Table 3: **Basic commands for working with files.** `$file` is a path you provide.

**Student Task 11:** The basic commands for working with files are listed in Table 3. Read their descriptions, you will need these commands to solve the following tasks.

**Student Task 12:** Change to `~/ex0` and find the first word of `sourcecode/debouncer.sv` with `cat`.

Your terminal might be too small to read the entire file. If so, try `less` instead. You can now scroll through the file line-by-line with the arrow keys, in half-screen jumps with `D` (“down”) and `U` (“up”), and in full-screen jumps with `Space` or `F` (“forward”) and `B` (“backward”). You can also jump to the end of the file with `Shift + G` and to the beginning with `G`. Finally, you can search with `/` followed by an expression terminated with `Enter`, and jump between matches with `N` (next match) and `Shift + N` (previous match). To quit `less`, press `Q` (“quit”).

Which third command can you use to view the beginning of that file?

**Sample solution:** `head sourcecode/debouncer.sv`. By default, `head` prints the first 10 lines, but this can be changed with the `-n` option, i.e., `head -n 42 $file`.

`head` and `tail` are very useful when you are only interested in the start or end of a file. `tail` is often used to inspect log files, when you are only interested in the latest entries.

**Student Task 13:** Find the last three entries of `examples/dmesg_log` with `tail` and note the statement:

**Sample solution:** `tail -n 3 examples/dmesg_log`

`examples/dmesg_log` is the log from a Linux kernel that has just started up (for demonstration purposes). You can use the same command line tools to figure out what went wrong when you experience crashes with other programs that generate log files.


In Unix systems, file endings like `.txt` bear no meaning other than being part of the file name. To create a plain text file, you can just create a file with any name.

**Student Task 14:** Create the new file `important_notes`. Figure out what command to use by looking at the command summary above and note the statement:

**Sample solution:** `touch important_notes`. (If the file already existed, `touch` would just have updated the access and modification time of the file without touching its content.)

### 3.3 Getting help

Usually, commands and programs you can execute in your terminal come with a manual. You will have to consult these often when using Unix systems, because most programs have an incredible amount of options. You could of course look up, e.g. the capabilities of `ls`, with a Google search. But its much faster to just use `man`, which will open its manual right in your terminal!

**Student Task 15:** Use the `man` command to view a program's manual. Of course, `man` can show you the manual of itself! `man` uses `less` to display a manual, so you can use the keys explained previously and quit with .

**Student Task 16:** In Unix systems, hidden files start with a `.` (period), like `.hidden_file`. When using `ls` without arguments, they will not be shown. Consult the manual of `ls` to find out how you can make it display hidden files as well. Are there any such hidden files in the `examples` folder?

**Sample solution:** Use the `-A` or `-a` option for `ls` to list also hidden files. `ls -a examples` shows that there is one such hidden file.

**Hint:** Use the `less` keys to search the manual page. The keyword you are looking for in this case is "ignore".

Some programs do not have a manual. However, they will (most likely) have a help page built in. You can usually access it by passing the `-h` or `--help` argument. For example, to see the short help page of `unzip` instead of the long manual, use:

```
sh> unzip --help
```

## 3.4 Terminal tips and tricks

There are a few things you really should know when working with a Linux terminal that will make your life a lot easier.

### 3.4.1 Tab completion

You might have noticed that typing in long paths is rather annoying, if not completely frustrating. You can make the shell complete file paths for you by pressing `Tab`.

Let us say you want to list the contents of `examples`. You start by typing:

```
sh> ls e
```

Pressing `Tab` will complete this to:

```
sh> ls examples/
```

If there is more than one option, nothing will happen. You can then press `Tab` again to get a list of possible completion options. Assuming you want to list the contents of a subdirectory of `~/ex0`, entering `ls` (with a space at the end) followed by `Tab Tab` will give you something like this:

```
calibre/  docs/      examples/  sourcecode/  
dfii/     encounter/ modelsim/  vivado/
```

**Note 3 (Copying and pasting):** You might have noticed the usual commands of `Ctrl + C` / `Ctrl + V` don't work in the terminal. This is because those keyboard commands already have another meaning (see Table 4).

To copy/paste from/to a terminal you can select any text with the left mouse button. The selection will automatically be placed in a special selection buffer. You can then paste it by pressing the middle mouse button (click the scroll wheel).

If that does not work or you prefer key combinations, most terminals support copying with `Ctrl + Shift + Up` + `C` and pasting with `Ctrl + Shift + Up` + `V`.

### 3.4.2 The history

The shell will keep a history of all commands you type in. You can scroll through the commands you typed by pressing `Up`. To get back again to more recent history, press `Down`.

But what if you know you used some command before, but just can't remember it? You can try to find it with the arrow keys, or just search for it by pressing `Ctrl + R`. To abort, press `Ctrl + C` or `Esc`.

If things get too cluttered, you can use the command `clear` to tidy up your terminal. This command clears the terminal screen, including its scroll-back buffers. If you want to clear the terminal screen but keep your scroll-back buffers, then you can use the `Ctrl + L` shortcut.



### 3.4.3 Keyboard commands

There are a few keyboard commands that can make you *a lot* faster when entering long commands. Instead of always using your arrow keys to edit commands, try some of these!















Keystroke	Description
 + 	Delete the part of the word before the cursor.
 + 	Delete the part of the line before the cursor.
 + 	Clear the terminal.
 + 	Go to the beginning of the line.
 + 	Go to the end of the line.
 + 	Terminate the currently running process.
 + 	End of file, quits the shell if pressed on an empty line.

Table 4: Useful keystrokes in the shell.

These tips and tricks are reprinted in the Cheat Sheet in Section 8 at the end of this exercise for your convenience.

## 4 Text Editors

At some point you will have to learn how to use a text editor. You might have worked with IDE's before or used GUI editors, like *Notepad++*. There are some very useful and extremely powerful tools available to you on the command line (e.g., *Vim* and *nano*) and also as graphical programs (e.g., *Emacs* and *Sublime-text*). The usual advice given to people new to Linux is: Try out *Vim* and *Emacs*, and learn to use one well.

### 4.1 Opening Graphical Text Editors

Many graphical text editors provide an environment that is familiar. Little to no learning is required (of course, there also exist very feature-rich graphical editors, e.g. *Sublime-text*). To open a graphical text editor from the command line, it is sufficient to type the name of the executable. For example, typing `gedit` opens *gedit*, the default text editor in the GNOME desktop environment. To open a given `$file` directly, enter:



```
sh> gedit $file
```

**Note 4 (Jobs in foreground and background):** Like pure shell programs (e.g., `less`), most GUI programs (e.g., `gedit`) “take over” the shell they were started in. This means that all input you type in the shell while the program is running go to that program instead of the underlying shell and that you only get output from that program. The program thus runs in *foreground*.

When starting a GUI application from the shell, this is most likely not what you want. In this case, you can start the program `$cmd` in *background* by appending a `&`:

```
sh> $cmd &
```

Programs running in background are not able to receive inputs through the shell but they can still print to the shell. This output can get mixed with output of other programs (even the one in foreground). Keep this in mind when seemingly unrelated messages pop up.

You can also put a program that is currently running in foreground to background: Hit  +  to suspend the program (causing the program to pause and all I/O, including its GUI, to freeze). You are now back to the underlying shell, where you enter `bg` (“background”) to continue the program in background. To put it back to foreground, enter `fg` (“foreground”).

It is possible to have multiple programs in background. You can get an overview of all programs started

from the current shell, which are called *jobs*, with `jobs`. The first number in the output table is the job ID. Enter

```
sh> fg %$jobid
```

to put the job with ID `$jobid` to foreground.

## 4.2 nano

*nano* is a very simple text editor for the command line that provides the most basic features that are expected from a text editor. It is useful for quick edits. *nano*'s main advantages are simplicity and ease of use. *nano* can be started by typing:

```
sh> nano $file
```

The most important shortcuts to know are `Ctrl` + `O` for saving the file, `Ctrl` + `X` to exit, and `Ctrl` + `G` for getting help.

By default, *nano* has some useful features disabled. It's a good idea to read the manual of *nano*, to get a feel of what *nano* can do. For example, you may want to enable mouse support by using the flag `-m`.

## 4.3 Vim

Vim has been a popular text editor for more than 25 years. It is very feature-rich and has a unique way of editing text that takes some time getting used to. Depending on how much typing you do, it will take a day or two until you are as fast as with *nano*. By far the best way to get started with *Vim* is to execute the command `vimtutor`. This provides a tutorial-style introduction.

To start Vim, type

```
sh> vim $file
```

Hit `I` to start inserting text. Once you are done typing, hit `Esc`. To save a file, make sure that you are **not** currently inserting text (i.e., press `Esc`, if you haven't already), then type `:w` and hit `Enter`. To quit, type `:q`, followed by `Enter` (again, make sure to hit `Esc` if you were inserting text before). Similarly, to quit without saving: `:q!`. Finally, to get help: `:help`. If something unexpected happens (which *will* happen, trust us), just hit `Esc`, followed by pressing `U` a couple of times. This will undo your last actions. To redo, hit `Esc` and then `Ctrl` + `R` to redo one step.

## 4.4 Emacs

GNU Emacs has been first released over 40 years ago. Just like Vim, it is a very powerful text editor, with a steep learning curve and a ton of features. It is popular in electronics engineering for its powerful *Verilog* and *VHDL* modes (*VHDL* mode is developed at ETH<sup>3</sup>). Use it with:

```
sh> emacs $file
```

To save a file, hit `Ctrl` + `X`, followed by `Ctrl` + `S` (you don't have to release the control key in-between). To close the editor, type `Ctrl` + `X`, followed by `Ctrl` + `C`. For getting help, there are various commands. Probably the most useful one is executed by hitting `Ctrl` + `H`, followed `A`. Then, type in the topic that you want to search for, and hit `Enter`. Emacs also has a tutorial that can be opened by `Ctrl` + `H`, followed by `T`<sup>4</sup>.

<sup>3</sup> <https://guest.iis.ee.ethz.ch/~zimmi/emacs/vhdl-mode.html>

<sup>4</sup> On the *tardis* machines, the tutorial is available by clicking on Emacs Tutorial after opening emacs.

## 4.5 Sublime-text

*Sublime-text* is a commercial lightweight graphical editor that is free to use and slightly more user-friendly than the alternatives above. It offers a variety of packages to expand its functionality, such as the *SystemVerilog* package to support SystemVerilog code we will be using throughout the exercises. If you are not that familiar yet with the numerous shortcuts in editors such as *Emacs*, *Sublime-text* can offer an easier start. *Sublime-text* can be started by typing:

```
sh> sublime_text $file
```

Note that entire directories can also be opened using sublime, simply replacing `$file` with a directory. While at home you can use PackageControl to install additional packages such as the SystemVerilog plugin, in the lab you will need to manually clone the package:

```
sh> cd ~/.config/sublime-text-3/Packages
sh> git clone https://github.com/TheClams/SystemVerilog
sh> cd ~/ex0/
```

**Student Task 17:** Try to do the following task with all the mentioned editors: Open `examples/edit_me.txt`, delete line 14, and save it.

**Hints:**

gedit	Use the mouse.
nano	Hit <code>[↓]</code> 14 times, then hold <code>[Del]</code> and hit <code>[Ctrl] + [O]</code> <code>[Enter]</code> <code>[Ctrl] + [X]</code> to save and quit.
vim	Enter <code>14G</code> to jump to line 14, enter <code>dd</code> to delete the entire line, and enter <code>:x</code> to save and quit.
Emacs	Hit <code>[Alt] + [G]</code> <code>[Alt] + [G]</code> 14 <code>[Enter]</code> to jump to line 14, <code>[Ctrl] + [Shift ↑] + [←]</code> to delete it, and finally <code>[Ctrl] + [X]</code> <code>[Ctrl] + [S]</code> to save and <code>[Ctrl] + [X]</code> <code>[Ctrl] + [C]</code> to quit.
sublime-text	Open the ex0 directory and use the mouse to select the file and edit.

## 5 Leveraging the Command Line

A purely text-based interface might initially come across as restricting and outdated. With the few basic concepts taught in this exercise, however, you will be able to use the shell to solve problems for which no single GUI tool exists.

### 5.1 Globbing (Wildcards)

Globbing is a useful tool to work with files that share a pattern. For example, to refer to all `.tc1` scripts in a directory, you can use `*.tc1`. The `*` means 'any number of any characters', so this pattern will give you all files that end with `.tc1`. Similarly, `directory/*` will give you *all* files in `directory`, *except* hidden files.

For example, to list all files ending in `.pp` inside the `encounter/src` directory, you could use:

```
sh> ls encounter/src/*.pp
```

There are a few more useful patterns, such as `?` which matches any *single* character, or `*.{tc1,pdf}`, which is a list that matches both `.tc1` and `.pdf` files. To learn more, you can have a look at `man 7 glob`.<sup>5</sup>

**Note 5:** The shell expands globs *before* executing a statement. Globs without matches, however, remain unchanged. In the example above, the statement that really gets executed is

```
sh> ls encounter/src/qfn40.VDD.pp encounter/src/qfn40.VSS.pp
```

<sup>5</sup> The '7' in `man 7 glob` designates Section 7 of the man pages. It must be specified for `glob` because there is a Linux function with the same name, which is described in Section 3. You can find an index of the sections of `man` in `man man`.

whereas

```
sh> ls encounter/src/*.cc
```

would be executed as-is because there are no files matching the glob.

If you want to prevent glob expansion, wrap the expression in quotes.

**Note 6 (Single vs. double quotes):** Enclosing characters in *single* quotes (') preserves the literal value of *each* character (including spaces and line breaks) within the quotes. *Double* quotes (") also preserve the literal value for all except the following characters, which get a special meaning: \$ and ` (see [Shell Expansions](#)), \ (see [Escape Character](#)), and \* and @ (see [Shell Parameter Expansion](#)).

Patterns for `find` or `grep` (described in the following sections), for example, should be enclosed in *single* quotes unless you need the special meaning of a character.

**Student Task 18:** Output the contents of all files that end in `.txt` in `examples/globbering/` to the command line with a *single* command. The output will tell you if you did it correctly. Note the statement you used.

**Sample solution:** `cat examples/globbering/*.txt`

**Hint:** Recall the `cat` program.

**Student Task 19:** Output the contents of all files that end in `.TEXT` and `.TXT` (mind the upper case letters!) in `examples/globbering/` to the command line with a *single* command. The output will tell you if you did it correctly. Note the statement you used.

**Sample solution:** `cat examples/globbering/*.T{E,}XT`, which is expanded to `cat examples/globbering/*.TEXT examples/globbering/*.TXT`. Note that the order of strings in the braces matters.

## 5.2 Pipes

Pipes are useful to connect multiple commands together. One of the design principles of Unix was “do one thing, and do it well”. This means, tools should be simple and powerful in their specific domain, but there should also be a mechanism to combine multiple programs in an easy way. This mechanism is the *pipe*, the `|` symbol. It allows you to connect the output of one program to the input of another program.

For example, we can combine `find` and `grep` to work together. Let's say you want to find all `.tcl` scripts for the *Encounter* tool. We could first find all `.tcl` files with `find`, and then `grep` for 'encounter':

```
sh> find . -name '*.tcl' | grep 'encounter'
```

You will find out exactly how `find` and `grep` work in the following sections.

**Student Task 20:** In Unix, you can use `wc` (wordcount) to count words, lines (`-l` flag) or characters of a file. With `cat`, you can output multiple files to the command line (concatenate). Use a pipeline to count the lines of all `.sv` files in `sourcecode/` (not including any subdirectories) and note the statement you used.

**Sample solution:** `cat sourcecode/*.sv | wc -l`

## 5.3 Sort

The `sort` command can be used to sort lines of text files. Of course the ability to sort something in some specific way can be useful in general, but it starts being especially useful when sorting inputs numerically. To sort numerically, `sort` makes use of the `-n` flag.

Another input type that often needs to be sorted is file sizes. For example, if we wanted to find out which of our directories used up the most space on our hard drive, we could use the `du` command in conjunction with the `sort` command. `du` stands for “disk usage”. Note that the `du` command provides information about disk usage, which differs from the file size. It reports in multiples of the filesystem block size, which means a file’s reported size may be larger if it doesn’t fully utilize a block. Block-oriented storage devices like hard drives and SSDs function by organizing data into fixed-size blocks. When a file is written to such a device, there might be a block that isn’t entirely filled. Consequently, disk usage, as shown by `du`, is rounded up to account for the entire block being used. The whole command looks as follows:

```
sh> du -sh * | sort -h
```

By now you should already know that `*` stands for “all files in the current directory”, or more precisely “all files with any number of characters in their name inside the current directory”. Perhaps you have noticed that in our example, both `du` and `sort` have the flag `-h`. Here, those two flags actually have a very similar meaning: In the case of `du`, it prints the file size “humanly readable”, i.e., “3.2K” instead of “3200”. In the case of `sort`, it *sorts* “humanly readable”, so that “3.2K” lands before “5.0M”.

In fact, it is common for many flags to mean roughly the same thing for a lot of different UNIX commands. For instance, the flag `--version` is implemented almost universally across hundreds of Linux commands, even ones that are not part of the standard UNIX commands. Another such flag is `-r`, which usually means “recursive” - though in `sort`’s case, it actually means “reverse”. These soft standards make learning the command line much easier, once you are familiar with the basics.

**Student Task 21:** Count the number of lines in each file in `sourcecode/`, and then sort them by the number of lines. Note the statement you used.

**Sample solution:** `wc -l sourcecode/* | sort -h`. The `-h` argument to `sort` is required to preserve the significance of the decimal digits when numbers are aligned with spaces instead of zeros (such as in the output of `wc`).

**Hint:** Take a look at Note 5 and check out the cheat sheet for useful commands.

**Student Task 22:** Order all files in `sourcecode/` by their size. The largest file should be at the top of the output. Remember that you can use `du` for tasks that involve outputting the size of files. Note the statement you used.

**Sample solution:** `du -b sourcecode/* | sort -h`. With the `-b` argument, `du` outputs the size of a file in bytes rather than the size of the filesystem blocks it occupies.

**Bonus:** Only output the three largest files.

**Sample solution:** `du -b sourcecode/* | sort -h | head -n 3`

**Hint:** Look at the manual of `head` and `tail`.

## 5.4 Find

Looking for specific files on your system is a task that one has to do fairly regularly. Two commands that are among the absolute best in looking for files with certain properties are `find` and `grep`. In this section, we cover `find`. As a rule of thumb, `find` is best used when you know that your file has certain metadata: maybe you know approximately when you have created the file, when you last looked at it, how large it is, etc. In contrast, `grep` is usually used when wanting to sift through *the content* of a file.

`find` works by essentially filtering all files through a chain of conditions. The conditions are given by flags. Working with `find` is fairly straightforward and almost like verbally explaining what you are looking for.

```
sh> find -mtime -1 -size +100k
```

This searches for all files that have been modified (`-mtime`) less than one day ago (`-1`) and have size (`-size`) larger than 100 Kilobytes (`+100k`). `find` has a lot of conditions where it makes sense to specify “greater than” or “smaller than”. This is specified by prefixing the amount with a `+` or a `-`. This is why the condition “larger than 100 Kilobytes” is written as `-size +100k`. If we additionally knew that our file was smaller than a gigabyte, we would type:

```
sh> find -mtime -1 -size +100k -size -2G
```

It’s necessary to use `-2G` here. This may seem a bit surprising at first, but this is just because file sizes are rounded up. Table 5 shows an overview of tests for `find` that are useful to know by heart.

Test	Description
<code>-name</code>	Matches the name of a file.
<code>-size</code>	Condition on the size of a file.
<code>-atime</code>	Condition on when file was last accessed.
<code>-mtime</code>	Condition on when file was last modified.
<code>-type</code>	Specifies type of file (directory, regular file...).

Table 5: Commonly used `find` tests.

**Student Task 23:** Search for all files in `~/ex0` with names ending in `.sv` that have a size larger than 5 Kilobytes and note the statement you used.

**Sample solution:** `find ~/ex0 -name '*.sv' -size +5k`

## 5.5 Grep and Regexes

While working with many files in different directories, you quickly find yourself losing overview. To quickly find something in one of many files, you can use `grep`. `grep` is a tool for searching a body of text to find a specified pattern. Those patterns are specified with a syntax called “Regular Expressions” or `regex(es)`.

Regexes can be very complicated (there are entire books on regexes). We’ll have a look at the very basics here. Regexes are comprised of a string of characters, e.g., `foo.bar[0-9]`. Let’s take that `regex` apart:

- `foo` matches the literal word “foo”,
- `.` matches any single character,
- `bar` matches the literal word “bar”, and
- `[0-9]` matches a single digit between 0 and 9.

That `regex` would match all of the following strings, among others: “foo\_bar1”, “foo bar0”, “fooobar9”, “foozbar5”, and “foolbar2”.

To quickly find out whether a string matches a `regex`, use `echo` to pipe the string into `grep`. For example:

```
sh> echo foolbar2 | grep -Eo 'foo.bar[0-9]'
```

would output the input string, `foolbar2`.

The `-o` and `-E` options are described together with other commonly used `grep` options in Table 6. Table 7 shows common `regex` pattern elements.

The syntax for `grep` is

```
sh> grep $options '$pattern' $path
```

`$options` and `$path` are optional. If you omit `$path`, `grep` searches the standard output (useful in pipes, like in the example above) or the current working directory when the `-r` option is used.

Command	Description
<code>-E</code>	Enable the extended regex syntax. This means that the characters <code>?</code> , <code>+</code> , <code> </code> , <code>{</code> , <code>}</code> , <code>(</code> , and <code>)</code> become special modifiers (see Table 7). If you want to match those characters literally, you have to escape them with a preceding backslash.
<code>-r</code>	Recursively look at all files under the given directory.
<code>-o</code>	Only print the matching strings, not the entire line.
<code>-h</code>	Do not print the filename, just the matching string/line.
<code>-l</code>	Only print the paths of matches, not the matched string.
<code>-n</code>	Prefix the output with the linenummer where the match occurred.

Table 6: Common options for **grep**.

Regex	Example	Description
MATCH EXPRESSIONS		
<code>&lt;char. literal&gt;</code>	<code>f</code>	Matches the single character <code>&lt;char. literal&gt;</code> literally. The example matches <code>f</code> .
<code>.</code>	<code>.</code>	Any single character.
<code>[&lt;char1&gt;-&lt;char2&gt;]</code>	<code>[a-z]</code>	A single character in the range from <code>&lt;char1&gt;</code> to <code>&lt;char2&gt;</code> (inclusive). The example matches any lowercase character. The character range can be specified multiple times, like so: <code>[a-zA-Z0-9]</code> . This would match any lower- or uppercase character or digit.
<code>[:alpha:]</code>	<code>idem</code>	A single letter.
<code>[:digit:]</code>	<code>idem</code>	A single digit.
<code>[:alnum:]</code>	<code>idem</code>	A single alphanumeric character (i.e., either letter or digit).
<code>[:xdigit:]</code>	<code>idem</code>	A single hexadecimal character.
<code>\w</code>	<code>idem</code>	A single “word” character, i.e., a digit, a letter, or an underscore.
<code>\.</code>	<code>idem</code>	A single period (i.e., <code>.</code> ) character. The literals <code>?</code> , <code>+</code> , <code> </code> , <code>{</code> , <code>}</code> , <code>(</code> , and <code>)</code> have to be entered correspondingly with a preceding backslash ( <code>\</code> ).
MODIFIERS		
<code>?</code>	<code>.*?</code>	The <code>?</code> means “maybe”, meaning that the preceding expression appears zero times or exactly once. The example would match an empty string or a string of length 1.
<code>*</code>	<code>.*</code>	The <code>*</code> means that the preceding expression appears zero, one, or many times. The example would match a string of arbitrary length, including an empty one. If multiple options for a match are available, the <code>*</code> is greedy and takes the longest one.
<code>+</code>	<code>.*+</code>	The <code>+</code> means that the preceding expression appears one, or many times. The example matches on a string of arbitrary length greater than one.
<code>{&lt;num&gt;}</code>	<code>{3}</code>	Matches on a string if the preceding expression appears exactly <code>&lt;num&gt;</code> times.
<code>{&lt;num1&gt;,&lt;num2&gt;}</code>	<code>{3,6}</code>	Matches on a string if the preceding expression appears between <code>&lt;num1&gt;</code> and <code>&lt;num2&gt;</code> times (inclusive).
SUBEXPRESSIONS		
<code>(&lt;pattern&gt;)</code>	<code>([:digit:]+\.)+</code>	Forms a subexpression. In the example, this is used to apply a modifier to a sequence of elements. The example matches one or multiple (outer <code>+</code> ) pairs of a number composed of one or multiple (inner <code>+</code> ) digits and a period.

Table 7: **Common regex pattern elements.** Words enclosed in `<` and `>` are used as placeholders.

**Student Task 24:** Try to find all files in `~/ex0` where the pattern ‘BUFMnW’, where `n` is a natural number (with possibly multiple digits, of course), occurs. Which `grep` options and what regex pattern do you have to use? How many unique matches are there? In which files are they?



**Sample solution:** The extended-syntax regex (`-E` flag to `grep`) we want to use is `BUFM[[:digit:]]+W`; in other words, match the string “BUFM” followed by one or many digits followed by a ‘W’. `grep -Eroh` (see Table 6 for an explanation of the options) and the expression above lists all matching words. Since there are duplicates, pipe that into `sort -u`. Pipe that result into a `wc -l` to count the number of unique words matching the pattern:

```
grep -Eroh 'BUFM[[:digit:]]+W' | sort -u | wc -l
```

Replace the `-oh` options to `grep` with `-l` to print the paths of the matching files instead:

```
grep -Erl 'BUFM[[:digit:]]+W'
```

**Hint:** Filter out duplicates in a newline-separated list by piping it into `sort -u` (for “unique”).

**Student Task 25:** Lots of people will leave their e-mail addresses in files. Try to find all ETHZ e-mail addresses (i.e., `user@department.ethz.ch` or `user@machine_or_group.department.ethz.ch`) in all files in the exercise directory. Come up with a regex to match e-mail addresses, and filter out any duplicates. How many can you find?

**Sample solution:** Assume that *department* and *machine\_or\_group* can contain alphanumeric characters, underscores, and dashes. The extended-syntax regex for this is `[[:alnum:]]\_\-]+`. *user* may additionally contain dots, so its regex is `[[:alnum:]]\.\_\-]+`. We can match e-mail addresses without *machine\_or\_group* with

```
[[:alnum:]]\.\_\-]+@[[:alnum:]]\_\-]+\.\ethz\.\ch
```

To include addresses with *machine\_or\_group*, we have to insert an optional subexpression (i.e., `<pattern>`) followed by a `?` between the ‘@’ and the *department* pattern. The pattern of the subexpression is that of the *department* derived above followed by a dot character. We thus obtain

```
[[:alnum:]]\.\_\-]+@[([[:alnum:]]\_\-]+\.)?[[:alnum:]]\_\-]+\.\ethz\.\ch
```

Piping this into `sort -u | wc -l` reveals that there are 4 e-mail addresses.

(As you can see, regexes can easily get quite complicated. In fact, deriving the appropriate regex can take even advanced users quite a bit of trial and error. We do not expect you to construct such complicated regexes at an exam.)

**Hint:** There are more than 0 but less than 5.

**Note 7:** If you want to test your regex commands on sample input, you can check out <https://regex101.com/> to evaluate your syntax.

## 5.6 Diff

When you want to compare two files, `diff` comes in handy. With `diff`, you can find the lines in which two files are different.

```
sh> diff file1 file2
```

**Student Task 26:** A friend of yours sent you a new version of a program you are working on. When testing it, you notice it doesn't work anymore (original: `examples/diff/prepare_chip.pl`, new: `examples/diff/prepare_chip_v2.pl`). Use `diff` to find out on which line your friend introduced the error.

**Sample solution:** There is a missing `"-1"` in line 269 of the `prepare_chip_v2.pl` file.

## 6 Bash Scripting Basics

### 6.1 Creating a simple bash script

In this task you will learn how to write a simple bash (shell) script. Bash scripts are extremely useful to automate repetitive tasks. They can make your life a lot easier and help you save time.

**Student Task 27:** We start by creating a script file and making it executable.

```
sh> touch ex0_script.sh
```

Now we have to make this executable. By default, a new file will not be marked executable. This is an important safety measure, imagine you could just execute any malicious script!

**Student Task 28:**

```
sh> chmod +x ex0_script.sh
```

`+x` means that executable rights are added. There are three basics permissions for files in Unix: *read*, *write* and *execute*. To see what rights are currently set, you can use `ls -l` (in many shell configurations abbreviated as `ll`). The leftmost column will indicate what rights the owner of the file, users from the same group as the owner, and everyone else has. The format can be observed in the two following examples: The first character indicates file type, then there are 3 `rwX` triplets for *owner*, *group* and *others*. For example:

```
-rw-r--r--
drwxr-xr-x
```

`r` stands for reading permission, `w` stands for writing permission, `x` stands for execution permission on files and access permission on directories, `d` means the file is a directory, and `-` indicates a normal file in the first column and "no right" otherwise.

**Student Task 29:** Let us continue the work on our script. Open `ex0_script.sh` in a text editor, and add the following as the first line: `#!/bin/bash`

This first line, starting with `#!`, (called "shebang" or "hashbang") tells the shell which program shall be used to interpret the rest of the file. In this case, we use Bash installed at `/bin/bash`.

To execute the script, do the following:

```
sh> ./ex0_script.sh
```

The `./` notation means, the shell will look to execute a file in the current directory. If you wouldn't specify this, the shell would go and search for a `ex0_script.sh` executable file in your default search path.

**Student Task 30:** You can now put anything in your script, that you could also type in the terminal by hand. Make your script output "Hello world!". **Hint:** Look up the `echo` command.

```
Sample solution: echo "Hello world!"
```

Bash is not entirely different from other programming languages you might have encountered. It also has control structures like for/while loops and if/else conditionals. The syntax for a simple if condition is as follows:

```
if [ "string1" != "string2" ]; then
    echo "the strings are not equal."
fi
```

**Note 8:** Contrary to most other programming languages, the spaces between the brackets and operators are *vital* in Bash. As a matter of fact, `[` is not actually part of the “bash language”, it is a separate program called `test`.

**Student Task 31:** Add an `if` condition to your script that tests if the file `examples/test_file` exists. If it does (it definitely should), output the message “file found”. **Hint:** Consult the manual of `test` (`man \ test`).

**Sample solution:**

```
if [ -e examples/test_file ]; then
    echo "file found"
fi
```

## 7 Bonus Section: Remote Access

Most computers can be accessed remotely through the command line. This allows for direct access to the computer's command line from a different machine, through a Secure Shell (`ssh`). This type of access does require the corresponding server to be running on the host computer (usually enabled on Linux systems), the correct ports being open for access through the network, and correct credentials, such as user and password information.

To access a remote machine, the client computer will need to start the communication from its own command line or corresponding program. As each OS is different, a list is provided below:

- **Linux:** Terminal
- **macOS:** Terminal
- **Windows:** PowerShell (for Windows versions above 10.1803)
- **Mobile OSes:** There exist a variety of Apps that allow you to access `ssh` for both iOS, Android, and similar, however a proper computer is recommended.

Once in the corresponding program, accessing a remote computer is as easy as typing the command `ssh` along with the username at the destination/host computer, the `'@'` symbol, and the computer's IP address or hostname. For computers at D-ITET, all hostnames are of the form `computer_name.ee.ethz.ch`. The command line should then prompt you for a password for the corresponding user. Please note that you may need to be within the ETHZ network (or VPN) for this.

```
sh> ssh $USER@$HOSTNAME
```

**Note 9:** Please be aware that typing passwords in the command line will not show up, not even showing a symbol for each character typed. You are typing, however, press enter to submit the password.

If you are outside of the ETHZ network and do not want to use a VPN, you have the possibility to establish a “direct” connection to the target host via the login server `login.ee.ethz.ch`.

```
sh> ssh -o ProxyJump=<Your ETH username>@login.ee.ethz.ch $USER@$HOSTNAME.ee\
.ethz.ch
```

**Student Task 32:** From your private machine, attempt to connect to the VLSI account on the machine you are using during the exercise.

As typing passwords can be quite cumbersome, authentication is also possible through an ssh key stored on your computer. These use public key cryptography to ensure the connecting client is authorized. To enable this, an ssh key needs to be generated and transmitted to the server.

If you have already generated an ssh key, for example to easily connect to github, this step may not be necessary. Generally, keys are stored in the hidden directory in your home folder `~/.ssh/` as a private and public pair. To generate a new key, use the `ssh-keygen` command below, substituting your email address. To copy the public key to a remote server, you can copy the contents of the `id_rsa.pub` file, or use the `ssh-copy-id` command below, which copies the content of your `id_rsa.pub` into a new line in `~/.ssh/authorized_keys` on the remote server.

```
sh> ssh-keygen -t rsa -b 4096 -C "your_email@example.com"
sh> ssh-copy-id $USER@$HOSTNAME
```

Copying files through the network is also possible using the Secure copy protocol (`scp`). The command is shown below and works similar to the `ssh` command regarding a host, while similar flags can be used as with the `cp` command, such as the `-r` flag for recursively copying a directory. Please note that in order to copy a file from your computer to the remote machine, the order can be reversed to the command below.

```
sh> scp $USER@$HOSTNAME:/path/to/source_file /path/to/target_file
```

**Note 10:** For a more scalable alternative to `scp` you can check out `rsync`.

If a GUI is desired from a remote computer, `vncserver` can be used. More information is available here: <https://computing.ee.ethz.ch/RemoteAccess/VNC>

## 8 Cheat Sheet

Command	Description
GETTING HELP	
man	Show the manual of a program, syscall, library, etc.
NAVIGATING	
pwd	Display the current working directory.
cd	Change the current working directory to the given directory.
ls	List all files and directories in the current working directory.
tree	Get a visualization of the directory tree under the current working directory.
VIEWING FILES AND DIRECTORIES	
cat	Concatenate the content of one or multiple files and output the result.
head	Output the first couple of lines to the terminal.
tail	Output the last couple of lines to the terminal.
less	Browse a file in a visual viewer.
diff	Compare files line by line.
MANIPULATING FILES AND DIRECTORIES	
mkdir	Create a new directory with a given name.
rmdir	Remove a directory. Will not work with non-empty directories.
rm	Remove a file or directory (recursively, with <code>-r</code> ) and its contents. <b>Warning:</b> There is no trashcan on the command line! Files and directories will be deleted irrevocably.
cp	Copy a source file or directory (recursively, with <code>-r</code> ) to a target.
mv	Move a file or directory. Also used to rename files/directories.
touch	Create a new file if it doesn't exist yet.
chmod	Change file permissions.
SEARCHING FILES OR CONTENT AND SORTING	
find	Search for files (by attributes) in a directory hierarchy.
grep	Find lines in files that match some pattern.
sort	Sort input by some criterion
VARIOUS UTILITIES	
du	Show file space usage.
uniq	Report or omit repeated lines.
wc	Count words, lines or characters.
REMOTE ACCESS	
ssh	Secure Shell to access the command line of a remote computer
scp	Secure Copy, to copy files from a remote machine

Table 8: Overview of important shell commands.

### History

The shell will keep a history of all commands you type in. You can scroll through the commands you typed by pressing `↑`. To get back again to more recent history, press `↓`.















But what if you know you used some command before, but just can't remember it? You can try to find it with the arrow keys, or just search for it by pressing `Ctrl + R`. To abort, press `Ctrl + C` or `Esc`.





If things get too cluttered, you can use the command `clear` to tidy up your terminal. This command clears the terminal screen, including its scroll-back buffers. If you want to clear the terminal screen but keep your scroll-back buffers, then you can use the `Ctrl + L` shortcut.

— Please also see flipside. —







## Keyboard commands

There are a few keyboard commands that can make you *a lot* faster when entering long commands. Instead of always using your arrow keys to edit commands, try some of these!

Keystroke	Description
 + 	Delete the part of the word before the cursor.
 + 	Delete the part of the line before the cursor.
 + 	Clear the terminal.
 + 	Go to the beginning of the line.
 + 	Go to the end of the line.
 + 	Terminate the currently running process.
 + 	End of file, quits the shell if pressed on an empty line.

**Note (Copying and pasting):** You might have noticed the usual commands of  +  /  +  don't work in the terminal. This is because those keyboard commands already have another meaning (see the table above).

To copy/paste from/to a terminal you can select any text with the left mouse button. The selection will automatically be placed in a special selection buffer. You can then paste it by pressing the middle mouse button (click the scroll wheel).

If that does not work or you prefer key combinations, most terminals support copying with  +  +  and pasting with  +  + .

## 9 Epilog



This exercise sheet was co-authored by TheAlternative<sup>6</sup> a community dedicated to Linux and Free Software in general. If you want to learn more about how you can use Linux, have a look at the courses and workshops we do during the *Linux Days* on their website.

If you are interested in more resources, check out the following links:

- **Bash Quick Guide:** <https://thealternative.ch/guides/bash.php>
- **Bash Hackers Wiki:** <http://wiki.bash-hackers.org/>

---

<sup>6</sup> <https://thealternative.ch/>