CN/DCI/163

Shell Support

or

tcsh and **zsh** for pedestrians

Arnaud Taddei *

September 6, 1995

Abstract

This document describes how to use the main features of `tcsh` and `zsh` for interactive work (not for shell programming). This paper shows differences and similarities of these two advanced shells.

*CERN - Arnaud.Taddei@cern.ch

Contents

1 Introduction **5**

1.1 Overview . . . 5

1.2 Conventions . . . 5

1.3 Remember the man pages . . . 6

2 General technical issues with shells **7**

2.1 Invocation of the shell . . . 7

2.1.1 Interactive login state . . . 7

2.1.2 Interactive non-login state . . . 7

2.1.3 Non-interactive or batch state . . . 7

2.1.4 Logout state . . . 8

2.2 Quotes . . . 8

3 Configurability **10**

3.1 Warning . . . 10

3.2 Builtin functions . . . 10

3.3 Variables . . . 10

3.3.1 Shell variables . . . 10

3.3.2 Environment variables . . . 11

3.4 Aliases . . . 12

3.5 Functions . . . 12

3.6 Options . . . 14

3.6.1 **tcsh** specific . . . 14

3.6.2 **zsh** specific . . . 14

3.7 User and global startup-files . . . 15

3.7.1 **tcsh** specific . . . 15

3.7.2 **zsh** specific . . . 15

3.8 Execution of other files and *source* of files . . . 16

4 Interactive functionalities **17**

4.1 Execution of commands . . . 17

4.2 Completion (or name-completion . . . 17

4.3 Multiline editing . . . 18

4.4 Name generation or *globbing* . . . 20

4.4.1 **zsh** specific . . . 21

4.5 History mechanism . . . 21

4.6 Redirections and filterings (pipes) . . . 24

Redirections . . . 24

Pipes . . . 25

4.6.1 **zsh** specific . . . 26

4.7 Spelling correction . . . 27

4.8 Setting the prompt . . . 28

4.9 Job control . . . 30

4.9.1 **zsh specific** . . . 30

5 Conclusion **31**

Appendix A - `tcsh` completion **32**

Appendix B - `csh` way for history mechanism used by `tcsh` and `zsh` **33**

Appendix C - `tcsh` environment variables **34**

Appendix D - `tcsh` shell variables **35**

Appendix E - `zsh` options **37**

Appendix F - `zsh` parameters set automatically **39**

List of Tables

1 Conventions 5
2 List of global input files for **tcsh** 15
3 List of user input files for **tcsh** 15
4 List of global input files for **zsh** 15
5 List of user input files for **zsh** 16
7 List of redirection operators in **tcsh** 24
9 List of redirection operators in **zsh** 27
10 List of prompt formats 29
11 Shell variables used by **tcsh** and involved in completion 32
How to use the history mechanism with **tcsh** 33
12 Summary of event designators 33

List of Examples

1 How to start an interactive shell 7
2 How to quote strings 8
3 How you can set shell variables 10
4 How you can check the content of a shell variable 10
5 How you can list shell variables 11
6 How you unset shell variables 11
7 How you can set environment variables 11
9 How you can check the content of an environment variable 11
10 How you can list environment variables 11
8 How to use the sequential operator ; 11
11 How you unset environment variables 12
12 How to set an alias 12
13 How to get the list of aliases 12
14 How to unset an alias 12
15 How you can define functions with **zsh** 13
16 How you can define functions with **zsh** 13
17 How you can define functions with **zsh** 13
18 How to set an option in **tcsh** 14
19 How to set an option in **tcsh** 14

20 How you can set options with **zsh** 14
21 How you can list options with **zsh** 14
22 How you can unset options with **zsh** 14
23 How to *source* another file 16
24 How to execute a program 16
25 How to use *TAB* key and what happens in the case of partial completion for filenames 17
26 Completion of directories in the context of a `cd` 18
27 Completion of environment and internal variables 18
28 *emacs* versus *vi* binding 18
29 How to go to the beginning of a line with `tcsh` 19
30 How to go to the end of a line with `tcsh` 19
31 Classical examples of name generation 20
32 How to use *Ctrl Xg* or *Ctrl X** 21
33 How to customize the behaviour of the history mechanism 22
34 How to get the history 22
35 How to use the $\uparrow$ key 22
36 How to use *Page Up* 23
37 How to use *Page Up* more efficiently 23
38 How to use *pipes* 25
39 How to use *tees* 26
40 Generalized redirections in **zsh** 26
41 How to use the spelling correction operator *ESC s* 27
42 How to correct a full line with `tcsh` 28
43 How to use the `correct` variable with `tcsh` 28
44 How to use the `correctall` option with `zsh` 28
45 How to change the *prompt* with `tcsh` 28
46 How to deal use `jobs` in `zsh` 30
47 How to use the `nohup` option with `zsh` 30
48 How to use the history mechanism with `tcsh` 33

1 Introduction

1.1 Overview

A shell-program (simply called *shell*) is a program which is the interface between the operating system and you, the user. It enables you to access system resources. In this paper only **tcsh** and **zsh** will be considered.

These two shells have advanced functionalities built into them such as completion, multiline editing, history mechanism. They provide mechanisms which really improve your efficiency in comparison with the features provided by the old shells like **sh** or **csh**.

Other shells are briefly described and compared in "A Shell Comparison" by Arnaud Taddei, reference 162, available from the Self Service shelves of the UCO in Building 513.

While reading this document you will notice that the behaviour of the shell and the behaviour of other programs can be affected by shell configuration files. A default for these file is usually provided by the system administrators [1] .

1.2 Conventions

Assumption:

It is assumed you logged into a machine named PLUS1. So after you logged in, wrote your password, you could be in the following state:

The prompt you get	Meaning in this document
[plus1] ~ >	To give an example which is common for **tcsh** and **zsh**
[tcsh] ~ >	To give an example which applies for **tcsh** only
[zsh] ~ %	To give an example which applies for **zsh** only

Conventions:

Table 1: Conventions

Key	Refers to
TAB	the Tab key on the keyboard.
ESC	the Esc key on the keyboard.
$\hookleftarrow$	the Enter or Return key on the keyboard.
$\uparrow$	the up arrow key on the keyboard.

[1] At CERN these configuration files are named HEPiX startup files. For further reading the document "Proposal for a Common Script and X Environment at DESY and CERN" by Axel Kohler and Arnaud Taddei, reference 168, is also available from the Self Service shelves of the UCO and explains how we are trying to homogeneise these configuration files across the different sites in such a way that you have the same environment in different sites.

1.3 Remember the man pages

This paper is not meant to replace the man pages. If you need detailed information, at any time run `man tcsh` or `zsh` according to your shell.

2 General technical issues with shells

2.1 Invocation of the shell

The shell can be invoked into 3 different states:

1 – interactive login
2 – interactive non-login
3 – non-interactive = batch

and can exit in one state:

4 – logout

2.1.1 Interactive login state

If the first character of argument zero passed to the shell is '-' then the shell is assumed to be a login shell.

This is typically the case after a **`telnet`**, a **`login`**, a **`rlogin`**, etc.
Another example is when you run an **`xterm -ls`**, then the shell in the **`xterm`** is a login shell.

2.1.2 Interactive non-login state

An interactive shell is one whose standard input and output are both connected to terminals. Thus an interactive shell can be a login shell.

This is typically the case when you run the shell while you are already in a shell.

plus1 ~ > **`tcsh`** ↩

Then you will enter in a new **`tcsh`** shell which will be interactive non-login.
This is what happens as well when you start an **`xterm`** without the **`-ls`** option.

2.1.3 Non-interactive or batch state

A non-interactive shell or batch shell is one where neither standard input or output are connected to terminals[2].

It is when you run an executable file which starts with **`#!/usr/local/bin/zsh`** for instance. Then the process spawns a **`zsh`** process which will be the interpretor for the rest of the executable file [3].

[2] Thus `BATCH` doesn't imply `NQS`

[3] It is also what happen after an `xrsh dxcern xterm -ls`. The xrsh starts a shell in batch mode. This shell starts the `xterm` client and the `xterm` client starts a shell inside it in a login state because of the `-ls` option.

2.1.4 Logout state

When the shell exists it is in logout state and can read some particular files (refer to 3.7).

2.2 Quotes

A character is said *quoted* when it stands for itself. All characters enclosed between a pair of single quotes (' ') are quoted. Enclosing strings in ' ' prevents any further interpretation. A single quote cannot appear within single quotes.

Inside double quotes (" "), parameter and command substitution occurs. The *interpolation* is enabled which means that the evaluation is performed inside the quotes (" ").

Enclosing strings in ` ` executes the string enclosed.

For example,

Let's set the variable A to the value `Hello`

```
[tcsh] ~ > set A = 'Hello' ↩
[zsh] ~ % A='Hello' ↩
```

Let's display the variable `A`

```
[plus1] ~ > echo $A ↩
Hello
```

Let's set the variable `B`

```
[tcsh] ~ > set B = 'say $A' ↩
[zsh] ~ % B='say $A' ↩
```

If you display it you will not get `Hello` but

```
[plus1] ~ > echo $B ↩
say $A
```

Instead if you want to `say Hello` where `say` is just the string `say`.

```
[tcsh] ~ > set C = "say $A" ↩
[zsh] ~ % C="say $A" ↩
```

Then the interpolation is enabled and when you display it:

```
[plus1] ~ > echo $C ↩
say Hello
```

Be careful when you use ` it will evaluate and execute what is inside:

```
[tcsh] ~ > set D = `date` ↩
[zsh] ~ % D=`date` ↩
```

When you display it you get (in my case):

```
[plus1] ~ > echo $D ↩
Tue Sep 27 09:15:51 MET 1994
```

which is the date when you entered `set D = `date`` or `D=`date`` and not when you `echo $D`. Note the way this is evaluated.

3 Configurability

3.1 Warning

You can customize almost any aspect of shell behaviour BUT please remeber that setting extra options (or combining) may cause other programs to behave unexpectedly.

3.2 Builtin functions

Some functions are *builtin* into the shell itself for both `tcsh` and `zsh`. It means that at any time you can execute it.

For example, `source`, `history`, `bindkey`, `set`, `setenv`, `setopt`, `function`, etc. are all builtin functions.

They allow you to set your environment (see 3.3, 3.3.1, 3.3.2) to define some shortcuts (see 3.4) or functions (see 3.5), execute other files (see 3.8).

Sometimes you are using them without knowing it. When you are navigating into your history(see 4.5) with the arrow keys, the keys have been bound to some functions, when you are completing some files (see 4.2) the completion mechanism is using the `compctl` builtin function, and so on for all the features (see 4).

3.3 Variables

There are two classes of variables:

shell variables (or internal variables)	–	These are only used by the shell. Usually they are written in lowercase by convention in `tcsh` but they can be written in uppercase in `zsh`.
environment variables	–	These are used by the shell and can also be used by other programs.

3.3.1 Shell variables

You can set shell variables in the following way:

```
tcsh ~ > set my_variable=value ↩
zsh ~ % my_variable=value ↩
```

If you want to check the current state of the variable:

```
plus1 ~ > echo $my_variable ↩
value
```

If you want to list all the shell variables do:

```
[plus1] ~ > set ↩
```

Note that they are given in alphabetic order for both **tcsh** and **zsh**. The difference in the output is that in **tcsh** you get only the shell variables;
zsh lists the environment variables as well.

If you want to unset a variable you simply run:

```
[plus1] ~ > unset my_variable ↩
```

3.3.2 Environment variables

You can set environment variables in the following way:

```
[tcsh] ~ > setenv MY_VARIABLE value ↩
[zsh] ~ % set MY_VARIABLE="value"; export MY_VARIABLE ↩ [4]
```
or
```
[zsh] ~ % export MY_VARIABLE="value"  ↩
```

> Note for **zsh**: Be careful not to put *spaces* around the = symbol!
>
> The second way of setting the environment variables in **zsh** is less "Bourne" so it will be less portable than the first way of setting them.

If you want to look at the current value of the variable (refer 3.3.1):

```
[plus1] ~ > echo $MY_VARIABLE ↩
 value
```

If you want to list all the environment variables which are set and what are their value, you can simply run:

```
[plus1] ~ > env ↩
```

[4] The symbol ; is the *sequential* operator. Thus the statement:

```
[plus1] ~ > cmd1 ; cmd2 ↩
```

is equivalent to:
```
[plus1] ~ > cmd1 ↩
[plus1] ~ > cmd2 ↩
```

and means that the shell will execute (or interpret) cmd1 and then cmd2 in this sequential order.

If you want to unset an environment variable you do:

```
[tcsh] ~ > unsetenv MY_VARIABLE ↩
[zsh] ~ % unset MY_VARIABLE ↩
```

3.4 Aliases

The shell maintains a list of *aliases* which you can define or redefine. An *alias* is the name for a string of characters. This string can of course be a command even a complex one including pipes, variables, etc, quoted or not quoted (refer to 2.2). The two built-in commands **alias** and **unalias** can set, list or unset aliases.

Alias substitution is done on the shell input before any other substitution except history substitution.

For `tcsh`, the generic form is:

```
alias alias-name ['|"]command["|']
```

For **zsh**, the generic form is:

```
alias alias-name=['|"]command["|']
```

For example, if you want to set an alias:

```
[tcsh] ~ > alias h 'history' ↩
```

```
[zsh] ~ % alias h='history' ↩
```

To list the existing aliases:

```
[plus1] ~ > alias ↩
```

To unset one alias:

```
[plus1] ~ > unalias h ↩
```

3.5 Functions

In **zsh** only you can set shell *functions*. This is not possible in `tcsh`.

Note, `function` is a builtin function of **zsh** (see 3.2).

Thus, you can set a function by:

```
zsh ~ % function H ↩
 >       history | grep ${*} ↩
zsh ~ %
```

Then if you want to use it you can write:

```
zsh ~ % H ls ↩
```

And you will get the entries in your history where the string `ls` appears.

You can unset a function with the **`unfunction`** builtin function of **`zsh`**.

```
zsh ~ % unfunction H ↩
```

You can list all the function definitions with:

```
zsh ~ % functions ↩
```

which is a builtin function of **`zsh`**.

`zsh` allows you to create your own commands by defining shell functions with the "Bourne" syntax as well:

```
zsh ~ % H () { ↩
 >       history | grep ${*} ↩
 > } ↩
zsh ~ %
```

Then if you want to use it you can write:

```
zsh ~ % H ls ↩
```

And you will get the entries in your history where the string `ls` appears.

Be careful there are some differences between functions and aliases. To illustrate them, you can use the following alias:

```
zsh ~ %  alias history=echo ↩
```

But be careful that aliases are expanded when the function definition is parsed, not when the function is executed (*static* vs *dynamic* link in terms of functional programming). It means that if now you run:

```
zsh ~ % H taddei ↩
```

you will not execute `echo` but the `history` itself.

3.6 Options

With some Bourne shells appeared the feature of *options*. An option is basically a shell-variable which can be switched `on` or `off`.

They affect the behaviour of the shell itself. The most common ones are `ignoreeof`, `noclobber`, etc.

This feature is not that explicit in `tcsh` but we can consider that `tcsh` has the same feature.

3.6.1 tcsh specific

In `tcsh`, some variables don't need any value when they are set. For example:

```
[tcsh] ~ > set ignoreeof ↩
```

So options can be set in the same way as shell-variables (refer 3.3.1).

And they also can be unset in the same way:

```
[tcsh] ~ > unset ignoreeof ↩
```

3.6.2 zsh specific

In `zsh` the options are case insensitive and underscores are ignored, that is, `allexport` is equivalent to `A__lleXP_ort`.

In `zsh` , you can set options in the following way:

```
[zsh] ~ % setopt optionname ↩
```

To list the options, do:

```
[zsh] ~ % setopt  ↩
```

Note: The format of the output itself will be different if the option `kshoptionprint` is set or not.

To unset options, do:

```
[zsh] ~ % unsetopt optionname ↩
```

Note: Case is ignored in option names, as are embedded underscores.

3.7 User and global startup-files

3.7.1 `tcsh` specific

If the following system startup files exist they are read before any user startup files:

Table 2: List of global input files for `tcsh`

File name	**Function**
`/etc/csh.cshrc`	This file is always read by `csh` or `tcsh`
`/etc/csh.login`	This file is read only at login time by either `csh` or `tcsh`
`/etc/logout`	This file is read by either `csh` or `tcsh` at logout time

`tcsh` looks for user startup files that allow you to set your own environment and preferences in your home directory. The files are:

Table 3: List of user input files for `tcsh`

File name	**Function**
`.tcshrc`	This file is executed at any time and `.cshrc` is not executed if `.tcshrc` exists. The execution can be avoided if you run `tcsh` with a `-f` option.It should contain commands to set up aliases, key bindings, variables and no commands that produce output or assume the shell is attached to a terminal.
`.cshrc`	This file is executed only if `.tcshrc` doesn't exist.
`.login`	This file is executed only at login time. This file should contain terminal settings.
`.logout`	This file is executed at logout time.

3.7.2 `zsh` specific

If the following system startup files exist they are read before any user startup files:

Table 4: List of global input files for `zsh`

File name	**Function**
`/etc/zshenv`	This file is read by `zsh` . It can be avoided if you run `zsh` with a `-f` option. This file should not contain commands that produce output or assume the shell is attached to a `tty`.
`/etc/zprofile`	This file is read only at login time.
`/etc/zshrc`	This file is sourced in interactive shells. It should contain commands to set up alias, functions, options, key bindings, etc.
`/etc/zlogout`	This file is read at logout time

`zsh` looks for user startup files that allow you to set your own environment and preferences in **`$ZDOTDIR`** or, if thi is not set, in your home directory. **`$ZDOTDIR`** is not defined by CERN provided scripts so these files are in your home directory. The files are:

Table 5: List of user input files for zsh

File name	Function
.zshenv	This file is executed by zsh. The execution can be avoided if you run zsh with a -f option. This file should not contain commands that produce output or assume the shell is attached to a terminal.
.zprofile	This file is executed only at login time.
.zshrc	This file is executed in interactive shells. It should contain commands to set up aliases, functions, options, key bindings, etc.
.zlogin	This file should contain only settings for the terminal type and runs some external commands (fortune, msgs, etc.)
.zlogout	This file is executed at logout time

3.8 Execution of other files and *source* of files

If you wrote a file containing *C-shell* code (for tcsh) or a *Bourne-Shell* code (for zsh) you can *source* it. It means that you will not start a new process but the shell will be able to read the file and to execute what is inside. Thus you will not *spawn* a new process.
To source a file:

```
[tcsh] ~ > source my_file ↩
[zsh] ~ % . my_file ↩
```

Notice the '.'.

If you wrote a file containing *C-shell* code with #!/usr/local/bin/tcsh at its begining or *Bourne-Shell* code with #!/usr/local/bin/zsh at its begining you can execute it and thus you will start a new process:

To execute a file my_executable_file:

```
[plus1] ~ > my_executable_file ↩
```

4 Interactive functionalities

Each shell has many features. Some of the more useful ones are shown here and the similarities between `tcsh` and `zsh` highlighted. The man pages have a complete list of the features of the shell you use.

4.1 Execution of commands

If a command name contains no slashes, the shell attempts to locate it[5] If there exists a shell builtin by that name, the builtin is invoked.

Otherwise, the shell searches each element of path for a directory containing an executable file by that name. If the search is unsuccessful, the shell prints an error message and returns a nonzero exit status.

4.2 Completion (or name-completion

When entering commands, it is not necessary to type the complete name, a unique abbreviation is enough. When you type *TAB* it will complete the name for you, echoing the full name on the terminal and entering it into the edit buffer. If the string you typed doesn't match anything, the terminal bell will ring (unless the shell variable *nobeep* is set as referred in section 3.3 or unless you set "visual bell" in an xterm).

This completion mechanism works for file names, directory names, command names, shell variables and user names. You can control which files are matched during completion with the shell variables below. (the shell variable `fignore` may be set to a list of suffixes to be disregarded during the completion).

Now, let's follow the next example about the completion of a filename:

```
plus1 ~ > ls -l ↵
-rw-r--r--   1 taddei   c3          470 Jun 29 08:35 xsession.env
-rw-r--r--   1 taddei   c3        13234 Jun  8 18:02 my_data1
-rw-r--r--   1 taddei   c3        12332 Jun  9 18:02 my_data2
-rw-r--r--   1 taddei   c3         4396 Jun 29 17:41 my_data3
drwxr-xr-x   3 taddei   c3         2048 Sep 22 18:39 bin
-rw-r--r--   1 taddei   c3         4396 Jun 29 17:41 blabla
```

```
plus1 ~ > emacs xseTAB
```

will complete the name `xsession.env` for you with the cursor at the end of the command line.

```
plus1 ~ > emacs xsession.env █
```

Let's assume you want to edit the file `my_data1`. If you type:

```
plus1 ~ > emacs my_TAB
```

[5] In case of `zsh` if there exists a shell function by that name, the function is invoked.

Then the shell will produce a partial completion in the following way:

```
plus1 ~ > emacs my_data
my_data1     my_data2    my_data3
```

In addition different things can occur depending on the shell variables which are set. You can have:

1 – nothing if the variable *nobeep* is set,
2 – a beep if the variable *nobeep* is unset,
3 – a list of alternatives produced by the shell (see section 4.4).

Moreover, for `zsh` you can cycle through the alternatives while entering *TAB* n times on the keyboard. It will prompt you `my_data1`, then `my_data2`, `my_data3`, etc.

Now, if you want to change directory here is how you can use the *TAB* key.

plus1 ~ > `cd ~/b`*TAB*

Then the shell will produce:

plus1 ~ > `cd ~/bin`

and it will not try to match `blabla` which is a file, not a directory.

If you want to complete the name of an environment variable,

plus1 ~ > `echo $DIS`*TAB*

will produce:

plus1 ~ > `echo $DISPLAY`

4.3 Multiline editing

Commands that the user types in may be edited using either emacs style or vi style control characters. two sets of control characters that *emacs* or *vi* are providing. You can use the arrow keys and the function keys. In some cases (depending on the keyboard mapping) here you can also use the *Page Up* and *Page Down* keys.

`tcsh` and `zsh` both have a command named `bindkey` which enables you to redefine the behaviour of each key or to find out what any or all of the keys do.

plus1 ~ > `bind emacs` ↩ or plus1 ~ > `bindkey -e` ↩

will bind all the keys to the *emacs* interpretation of characters. Otherwise a:

plus1 ~ > `bind vi` ↩ or plus1 ~ > `bindkey -v` ↩

will bind all the keys to the *vi* interpretation of the charcaters.

Let's take an example. Assume you made a mistake in writing `/uxr` rather than `/usr`. The cursor is at the end of the command line. You could use the left arrow key to go back to the beginning of the command line but there is a more efficient way:

plus1 ~ > `/uxr/local/bin/emacs my_nice_file_with_a_long_name` █

emacs style	–	You can type the *Ctrl A* keys and the cursor █ will go to the begining of the command line because this sequence of keys does it in *emacs*.
vi style	–	You can type the ^ key and the cursor █ will go to the begining of the command line because this key does it in *vi*.

An then you will be in the following state.

plus1 ~ > `/uxr/local/bin/emacs my_nice_file_with_a_long_name`

Then, you just need to use the right arrow key → and change `x` to a beautiful `s`.

Let's go on with the same example but this time, assume the cursor is at the beginning of the command line and you would like to correct the upper case E:

plus1 ~ > `/usr/local/bin/emacs my_nice_file_with_a_long_namE`

emacs style	–	You can type the *Ctrl E* keys and the cursor █ will go to the end of the command line because this sequence of keys does it in *emacs*.
vi style	–	You can type the $ key and the cursor █ will go to the end of the command line because this key does it in *vi*.

An then you will be in the following state.

plus1 ~ > `/usr/local/bin/emacs my_nice_file_with_a_long_namE` █

You just need to use the left arrow key ← and change the `E` to a beautiful `e`.

Another interesting feature is the fact that you can edit multiple lines. Assume, your window is too narrow and you want to enter a very long command, then both `tcsh` and `zsh` are able to handle multiple line editing:

plus1 ~ > `/uxr/local/bin/emacs my_nice_file_with_`
`a_long_name` █

You can navigate with all the emacs or vi commands including with the arrow keys if they are available on your keyboard.

4.4 Name generation or *globbing*

All advanced shells provide a name generation or *globbing* mechanism. This feature enables you to produce automatically generated names. For instance assume you have the following in your current directory:

```
plus1 ~ > ls ↩
 OODIST    dec_a   lsof.8     mem.c  node.c  sock.o  user.c
 OOREADME  dev.c   lsof.h     mem.o  node.o  store.c user.o
 Makefile  dev.o   lsof230tar misc.c print.c store.o
 Makefiles fix.log machine.h  misc.o print.o sys
 arg.c     irix511 main.c     mnt.c  proto.h text.c
 arg.o     lsof    main.o     mnt.o  sock.c  text.o
```

If you want to get only the bodies of your C code:

```
plus1 ~ > ls *.c ↩
 arg.c   dev.c   main.c  mem.c   misc.c  mnt.c   node.c
 print.c sock.c  store.c text.c  user.c
```

If you want to get the bodies and the object code:

```
plus1 ~ > ls *.[co] ↩
 arg.c dev.c main.c mem.c misc.c mnt.c node.c print.c sock.c store.c
 text.c  user.c
 arg.o dev.o main.o mem.o misc.o mnt.o node.o print.o sock.o store.o
 text.o  user.o
```

If you want to get the files with the name `lsof.` with an unknown extension of 1 character:

```
plus1 ~ > ls lsof.? ↩
 lsof.8  lsof.h
```

But you may be want to get all the files but the files ending with `.c`:

```
plus1 ~ > ls *.[^c] ↩
 arg.o  lsof.h    mem.o  node.o  sock.o  user.o
 dev.o  machine.h misc.o print.o store.o
 lsof.8 main.o    mnt.o  proto.h text.o
```

Or you may want to produce the names of the files which do not end with `.o` or `.h`

```
plus1 ~ > ls *.[^oh] ↩
 arg.c   dev.c   lsof.8  main.c  mem.c   misc.c  mnt.c   node.c  print.c
 sock.c  store.c text.c  user.c
```

But you can also want to produce names while the cursor is on the command line in order to know, what names are available.
You can do it with *Ctrl D* keys like in:

```
[plus1] ~ > cd ~/docs/mCtrl D
```

will produce the whole list of files and directories under `~/docs` which start with 'm'.

> Note:
>
> ```
> [plus1] ~ > ~ Ctrl D
> ```
>
> will produce the list of all login names on the current system.
>
> ```
> [plus1] ~ > Spaces Ctrl D
> ```
>
> will produce the list of files under the current working directory.

Listing or expanding words that match a name containing wildcard characters can be done via the `list-glob` or `expand-glob` functions *Ctrl Xg* or *Ctrl X**

```
[plus1] ~ > ls ↩
 foo.c bar.c a.out

[plus1] ~ > ls *.cCtrl Xg
 foo.c bar.c

[plus1] ~ > ls *.cCtrl X*
[plus1] ~ > ls foo.c bar.c █
```

Command name recognition works also with the *TAB* or *Ctrl D* keys.

4.4.1 zsh specific

`zsh` has all the basic mechanisms described in 4.4 but some other features are provided.

- If the `extendedglob` option is set, it enables to reject some patterns,
- An expression of the form `<x-y>` matches a range of integers.
- The `numericglobsort` option will sort files with numbers according to the number. But this works with `echo`, not with `ls`.
- Grouping is possible.
- The string `**/` forces a recursive search of subdirectories.
- It is possible to exclude certain files using the `~` character.

4.5 History mechanism

The *history mechanism* is the mechanism by which the shell is able to keep in memory a sublist of the commands you entered. Each commands you entered receives a sequential number (and even a time stamp in the case of `tcsh`) and is kept in a memory stack. The length of this stack is controlled by some shell variables and the content of the stack is usually dumped onto a history file in your home directory (the default is `~/.history`) each time you logout of the shell. At login time, the shell reads the content of the history file and load it into memory. So it keeps some memory of your previous work. It's also why sometimes, you don't start your history stack with number one but with another number.

As the history mechanism keeps a track of the last commands you entered, you can recall them back either to reexecute them either to edit them and modify them. Let's assume you are producing a document using the LaTeXprocessor and you have already entered the following commands:

```
plus1 ~ > set history=6 ↩
...
plus1 ~ > emacs my_document.tex & ↩
...
plus1 ~ > latex my_document ↩
...
plus1 ~ > dvips -o my_document.ps my_document ↩
...
plus1 ~ > ghostview my_document.ps ↩
...
plus1 ~ > ls -l ↩
...
plus1 ~ > bibtex my_document ↩
...
```

Then your history will look like (be careful, the commands are not stamped by time with zsh):

```
plus1 ~ > history ↩
 210 18:05 emacs my_document.tex &
 211 18:15 latex my_document
 212 18:18 dvips -o my_document.ps my_document
 213 18:20 ghostview my_document.ps &
 214 18:21 ls -l
 215 18:23 bibtex my_document
```

Let's go on with this example.

Assume you want to re-run LaTeXbut you want to apply it to the file my_document1. Then you don't need to re-write:

```
plus1 ~ > latex my_document1 ↩
```

but you simply need to type five times the arrow key ↑ on your keyboard (if it exists of course) and you will get:

`plus1 ~ > bibtex my_document █`	after the first ↑
`plus1 ~ > ls -l █`	after the second ↑
`plus1 ~ > ghostview my_document.ps █`	after the third ↑
`plus1 ~ > dvips -o my_document.ps my_document █`	after the fourth ↑
`plus1 ~ > latex my_document █`	after the fifth ↑

Now you can press the return ↩ key or edit the command to process `my_document1` file instead of `my_document`.

Note that at each step the command can be edited as seen in 4.3.
It exists a better way for doing this using the *Page Up* key on your keyboard. This way is close to the `!str` event designator because you can call back the last command which starts with the `str` string. The difference is that you can edit the command. You can't do it with `!str`.

The *Page Up* key corresponds to the following escape sequence `^[p` (or *ESC p*) which is bound to the *history-search-backward* function.

The *Page Down* key corresponds to the following escape sequence
`^[n` (or *ESC n*) which is bound to the *history-search-forward* function.

So, if you want to re-rerun LaTeXfor `my_document1` you can do:

plus1 ~ > `l`*Page Up*

and you jump to
plus1 ~ > `ls -l` █

If you presse again *Page Up*, you receive:

plus1 ~ > `latex my_document` █

And you can edit the command line.

You can do it even faster with:

plus1 ~ > `la`*Page Up*

and jump directly to

plus1 ~ > `latex my_document` █

You can control the length of the history stack in memory by setting the `history` variable:

tcsh ~ > `set history=4` ↩
zsh ~% `set HISTSIZE=4` ↩

You can control the length of the history file by setting the `savehist` variable:

tcsh ~ > `set savehist=2` ↩
zsh ~% `set SAVEHIST=2` ↩

You can change the name of your history file by setting the `histfile` variable:

```
tcsh ~ > set histfile="~/.my_history" ↩
zsh ~% set HISTFILE="~/.my_history" ↩
```

then the following output should appear:

```
plus1 ~ > history ↩
 212 18:18 dvips -o my_document.ps my_document
 213 18:20 ghostview my_document.ps &
 214 18:21 ls -l
 215 18:23 bibtex my_document
```

because the shell keeps only the four last entries and if you log out now, your `~/.my_history` file will only contain:

```
 ls -l
 bibtex my_document
```

because the value of *savehist* is 2.

4.6 Redirections and filterings (pipes)

Like any other process in UNIX, the shell uses three so-called *file descriptors* numbered 0, 1 and 2 and also called *STDIN*, *STDOUT* and *STDERR* for *standard input*, *standard output*, *standard error-output*. *standard input* is where the process reads its input, *standard output* is where it writes its output and the *standard error-output* is where it writes error messages. By default, *filedescriptor* 0 (meaning, *standard input* is attached to the terminal keyboard and *filedescriptors* 1 and 2 are attached to the terminal screen.

Thus, you can *redirect* and/or *pipe* input and outputs of one process into other processes like you do with flows of fluid in pipes in a chemistry process!

Redirections

The following is the table which summarises all the possible redirection operators in `tcsh` (for `zsh` refer 4.6.1).

Table 7: List of redirection operators in `tcsh`

Operator	**Meaning**
`<`*word*	Open file *word* (which is first variable, command and filename expanded) as (to redirect) standard input to the process.
`>`*word*	Open file *word* as (to redirect) standard output. *word* can be something else (a file, a device, etc.)
`>!`*word*	Same as `>`, except that the file is truncated to zero length if it exists, even if the option *NO_CLOBBER* is set.
`>&`*word*	Open file *word* as (to redirect) standard output and standard error-output (the same as `>`*word* `2>&1`). *word* can be something else (a file, a device, etc.)

<table>
<tr><th>Operator</th><th>Meaning</th></tr>
<tr><td>>>&word</td><td>Appends the standard output and the standard error-output to word (a file, a device, etc.) (the same as >>word 2>&1).</td></tr>
<tr><td>>>word</td><td>Appends the standard ouptut to word (a file, a device, etc.)</td></tr>
<tr><td>>>!word</td><td>Same as >>, except that the file is created if it does not exist, even if the option NO_CLOBBER is set.</td></tr>
<tr><td><<<i>word</i></td><td>The shell input is read up to a line that is the same as <i>word</i>, or to an <i>End-Of-File.</i> No parameter substitution, command substitution or filename generation is performed on <i>word.</i> The resulting document, called a <i>here document</i>, becomes the standard input. If any character of <i>word</i> is quoted with single or double quotes or a \, no interpretation is placed upon the characters of the document. Otherwise, parameter and command substitution occurs, \ followed by a <i>newline</i> is removed, and \ must be used to quote the characters \, $, ‘, and the first character of <i>word.</i> If <<- is used, then all leading <i>tabs</i> are stripped from <i>word</i> and from the document.</td></tr>
<tr><td>|</td><td>Connect the standard output of one process to the standard input of another process.</td></tr>
<tr><td>|&</td><td>Connect the standard output and the standard error of one process to the standard input of another process.</td></tr>
</table>

The typical usage of redirections takes the form:

```
plus1 ~ > command [options and arguments] < input-file > output file ↩
```

Note: if `noclobber` is set, it places restrictions on output redirection to ensure that files are not accidentaly destroyed, and that >> redirections refer to exisiting files.

Pipes

The pipe operator is a binary operator represented by the symbol `|`.

It is a mechanism of communication between processes. It enables you to connect the standard output of a processus A to the standard input of a processus B. So, you can *plug* different processes together by creating pipes among them in a whole string.

As an example, you can try (just for fun):

```
plus1 ~ > ls -alg | egrep '^d' | egrep -v '\.$|\.\.$' | wc -l ↩
          24
```

The command `ls -alg` lists all the files and directories in the current directory in the following format:

```
drwx------  3 taddei   c3            512 Mar  9 11:26 News
-rwx------  3 taddei   c3            512 Mar  9 11:26 blabla
```

Each time the process `ls -alg` finds a file it writes it in this format to its standard output which is connected to the input of the second process named `egrep '^d'`.

For each line from the output of `ls -alg`, `egrep '^d'` tries to match `'^d'` regular expression: it will match only lines beginning with `d`.

Thus, `egrep '^d'` writes only the line begining by a 'd' to `egrep -v '\.$|\.\.$'`. `egrep -v '\.$|\.\.$'` will match lines which don't end with '.' or '..'.

Finally, `wc -l` counts the lines if any. Then, this long pipe calculates the number of non-trivial directories in the current directory.

But if you want as well to know the names of the non-trivial directories, you can add *tees* using the `tee` command in order to write the intermediate output to a file `ntd` building a complex *pipeline*. As an example,

```
[plus1] ~ > ls -alg | egrep '^d' | egrep -v '\.$|\.\.$' | tee ntd | wc -l ↩
        24
```

and you can find the full list of non-trivial directories in the file `ntd`.

4.6.1 zsh specific

Apart from all the regular redirections in `tcsh`, `zsh` can do more and table 4.6.1 lists all additional operators. Mainly, you can send the output of a process to more than one file, by specifying more redirections like:

```
[zsh] ~ % echo "Hello World" >file1 >file2 ↩
```

Which sends the output to the files `file1` and `file2`.

```
[zsh] ~ % make > make.log | grep Error ↩
```

Which sends the output to the file `make.log` and to the *pipe* to the `grep` command.

And this is the same for the standard input:

```
[zsh] ~ % sort <file1 <file2 <file3 ↩
```

which will correspond to a `cat` command:

```
[zsh] ~ % cat file1 file2 file3 | sort
```

In the next,

```
[zsh] ~ % cut -d: -f1 /etc/passwd | sort <newnames ↩
```

the `sort` process will read first the output of `cut` and then of `newnames`.

Table 9: List of redirection operators in zsh

Operator	Meaning
<*word*	Open file *word* as (to redirect) standard input to the process
>*word*	Open file *word* as (to redirect) standard output. *word* can be something else (a file, a device, etc.)
>!*word*	Same as >, except that the file is truncated to zero length if it exists, even if the option *NO_CLOBBER* is set.
>&*word*	Open file *word* as (to redirect) standard output and standard error-output (the same as >*word* 2>&1). *word* can be something else (a file, a device, etc.)
>>&*word*	Appends the standard output and the standard error-output to *word* (a file, a device, etc.) (the same as >>*word* 2>&1).
>>*word*	Appends the standard ouptut to *word* (a file, a device, etc.)
>>!*word*	Same as >>, except that the file is created if it does not exist, even if the option *NO_CLOBBER* is set.
<<[-]*word*	The shell input is read up to a line that is the same as *word*, or to an *End-Of-File*. No parameter substitution, command substitution or filename generation is performed on *word*. The resulting document, called a *here document*, becomes the standard input. If any character of *word* is quoted with single or double quotes or a \, no interpretation is placed upon the characters of the document. Otherwise, parameter and command substitution occurs, \ followed by a *newline* is removed, and \ must be used to quote the characters \, $, ', and the first character of *word*. If <<- is used, then all leading *tabs* are stripped from *word* and from the document.
<<<*word*	Open a file containing *word*, after expansion, as standard input.
<&*digit*	The standard input is duplicated from file descriptor *digit* (see dup(2)).
>&*digit*	The same as <& but for the standard output.
<&-	Close the standard input.
>&-	Close the standard output.
<&p	The input from the coprocess is moved to the standard input.
>&p	The output to the coprocess is moved to the standard output.
|	Connect the standard output of one process to the standard input of another process.

4.7 Spelling correction

The *spelling correction* feature helps you correct basic mistakes while writing a command.
The typical example is when you have made a mistake in a path name (refer 4.3):

```
plus1 ~ > /uxr/local/bin/emacs my_nice_file_with_a_long_name█
```

Rather than trying to go at the begining of the command line, at this
point you can press *ESC s* just behind the word which is mispelled and the system will try to correct the 'x' in 's' providing the following as the next step:

```
plus1 ~ > /uxr/local/bin/emacs{\it {ESC s}} my_nice_file_with_a_long_name
plus1 ~ > /usr/local/bin/emacs my_nice_file_with_a_long_name █
```

Note that it will correct the word immediately before the cursor (█).

Spelling correction of the entire command line (independantly of where is the cursor) can be achieved using *ESC* $. The shell will scan all words of the line (but the ones which starts with one of the following character: `!\^-*%`) and it will correct the command on the command line:

```
tcsh ~ > moree ~/Priject/SHELLS/sh3.texESC $
tcsh ~ > more ~/Project/SHELLS/sh3.tex█
```

It can be done as well using the `correct` variable (for `tcsh`) or option (for `zsh`). In this case `tcsh` will prompt you for its solution.

```
tcsh ~ > set correct=all ↩
tcsh ~ > moree ~/Priject/SHELLS/sh3.tex ↩
CORRECT>more ~/Project/SHELLS/sh3.tex (y|n|e)? █
```

Where `y` means of course `yes`, `n` means `no` and `e` means `edit`.

Answering `y` will execute the command `more ~/Project/SHELLS/sh3.tex`.

For `zsh` you can't use the *ESC* $ keys but you can set the `correct` or `correctall` options which will correct only the word before the cursor or the all line accordingly.

Thus,

```
zsh ~% setopt correctall ↩
zsh ~% moree ~/Priject/SHELLS/sh3.tex ↩
zsh: correct 'moree' to 'more' [nyae]? █
```

Then if you answer `y` you will have the next answer:

```
zsh: correct 'moree' to 'more' [nyae] y
zsh: correct '~/Priject/SHELLS/sh3.tex' to
'/afs/cern.ch/user/t/taddei/Project/SHELLS/sh3.tex' [nyae]? █
```

And if you answer `y` it will execute the command.

Note in this example, `a` means `abandon`.

4.8 Setting the prompt

The prompt is really a mather of taste and fortunately you can modify it as you want. Just for fun I will let you try this one:

```
tcsh ~ > set prompt='%m [%h] %S[%@]%s %B[%n]%b %U[%/]%u try to work %#' ↩
```

which produces for me:

```
plus1 [46] 18:44pm [taddei] [/afs/cern.ch/user/t/taddei] try to work > █
```

and in `zsh`

zsh ~ % PROMPT='%m [%h] %S[%@]%s %B[%n]%b %U[%/]%u try to work %#' ↩

which produces for me:

plus1 [46] 18:44pm [taddei] [/afs/cern.ch/user/t/taddei] try to work % ▮

The following table summarizes used formats in the example.

Table 10: List of prompt formats

Format	Meaning
`%m`	Machine name stopped at the first '.' charcater met. For instance it will write `dxcern` and not `dxcern.cern.ch`
`%M`	The full machine hostname.
`%h`, `%!`, `!`	Current history event number
`%S (%s)`	Start (stop) the reverse video mode (or standout mode)
`%@` or `%t`	Current time of day, in 12-hour format am/pm format.
`%T`	Current time of day, in 24-hour format.
`%p`	Current time of day, in 12-hour format am/pm format with seconds.
`%P`	Current time of day, in 24-hour format with seconds.
`%d`	The weekday in `<Day>` format.
`%D`	The day in `dd` format.
`%m`	The month in `<Month>` format.
`%W`	The month in `mm` format.
`%y`	The year in `yy` format.
`%Y`	The year in `yyyy` format.
`%l`	The line (`tty`) the user is logged on.
`%L`	Clear from prompt to the end of display or end of the line.
`%B (%b)`	Start (stop) the boldfacing mode. (Only if `tcsh` was compiled to eight bit clean)
`%n`	The user name
`%U (%u)`	Start (stop) the underline mode. (Only if `tcsh` was compiled to eight bit clean)
`%/`	Current working directory. See also `%~` `%c` `%C` in `tcsh` manual pages
`%~`	`cwd`. If it starts with $HOME, that part is replaced by a `~`. In addition if a directory name prefix matches a users's home directory, that part of the directory will be substituted with `~user`. NOTE: The `~user` substitution will only happen if the shell has performed a `~` expansion for that user name in this session.
`%c` or `%.`	Trailing component of `cwd`, may be followed by a digit to get more than one component, if it starts with $HOME, that part is replaced with a `~`.
`%C`	Trailing component of `cwd`, may be followed be a digit to get more than one component, no `~` substitution (only for `tcsh`).
`%#`	A `#` if the user is root and `>` otherwise.
`\c` or `^c`	'c' is parsed the same way as in `bindkey`.
`%%`	A single `%`.
`%?`	Return the code of the last command executed just before the prompt.
`%R`	In `prompt3` this is the corrected string; in `prompt2` it is the status of the parser.
`%{..%}`	Include string as a literal escape sequence. Note that the enclosed escape sequence, should only be used to change terminal attributes and should not move the cursor location. Also, this cannot be the last charcater in the prompt string (available only if the shell was compiled eight bit clean).

4.9 Job control

When a job is started asynchronously with `&`, the shell prints a line which looks like:

```
[1] 1234
```

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If you are running a job and wish to do something else you may hit the key `^Z` (control-Z) which sends a `TSTP` signal to the current job. The shell will then normally indicate that the job has been 'suspended', and print another prompt. You can then manipulate the state of this job, putting it in the background with the `bg` command, or run some other commands and then eventually bring the job back into the foreground with the foreground command `fg`. A `^Z` takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed.

A job being run in the background will suspend if it tries to read from the terminal.

The shell learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible.

4.9.1 zsh specific

If the `monitor` option is set, an interactive shell keeps a table of current jobs, printed by the `jobs` command, and assigns them small integer numbers.

```
zsh ~ % setopt monitor ↩
zsh ~ % xterm &↩
[1] 41989
zsh ~ % jobs↩
[1]  + running    xterm
zsh ~ %  unsetopt monitor↩
zsh ~ % jobs↩
```

There is no answer now.
When you try to leave the shell while jobs are running or suspended, you will be warned that 'You have suspended (running) jobs.' You may use the jobs command to see what they are. If you do this or immediately try to exit again, the shell will not warn you a second time; the suspended jobs will be terminated, and the running jobs will be sent a SIGHUP signal.

Assume `monitor` is set:

```
zsh ~ % setopt nohup ↩
zsh ~ % xterm & ↩
zsh ~ % exit ↩
zsh: you have running jobs.
zsh ~ % exit ↩
```

And now you left the shell.

5 Conclusion

This document described the main advanced features of two shells `tcsh` and `zsh`. Although they are from different backgrounds – `tcsh` from the C-shell family and `zsh` from the Bourne shell family - they have similar behaviours and even use the same keys and key bindings for the same functions.

Appendix A - tcsh completion

Table 11: Shell variables used by tcsh and involved in completion

Shell variable	Meaning
fignore	List of file name suffixes (e.g. .o, ~) to ignore during complete.
autolist	List possibilities on an ambiguous completion.
recexact	Recognize exact matches even if they are ambiguous.
autoexpand	Invoke the expand-history function automatically on completion.
nobeep	Disables *beeping* completetely.
matchbeep	Control *beeping* on completion. With matchbeep=nomatch, completion only beeps when there is no match, with matchbeep=ambiguous, beeps also when there are multiple matches, with matchbeep=notunique, beeps when thre is one exact and other longer matches, with matchbeep=never, it never beeps.

Appendix B - csh way for history mechanism used by tcsh and zsh

tcsh **specific**

Now if you want to call on command in this history, intcsh (and even in csh) you could have done:

Cmd	What to do	Result
The last cmd	plus1 ~ > !! ↩	plus1 ~ > bibtex my_document ↩
ghostview	plus1 ~ > !213 ↩	plus1 ~ > ghostview my_document.ps ↩
ls -l	plus1 ~ > !l ↩	plus1 ~ > ls -l ↩
latex	plus1 ~ > !la ↩	plus1 ~ > latex my_document ↩

But, bee careful that the shell doesn't give you the cursor, so you can't edit the line and it is executed. See below how to use the arrows in other to be able to edit the previous command.

The following table summarizes the event designators.

Table 12: Summary of event designators

Event designator	Meaning
!	Start a history substitution, except when followed by a *blank*, *newline*, = or (.
!!	Refer to the previous command. By itself, this substitution repeats the previous command.
!n	Refer to command line n. This is an absolute referencing.
!-n	Refer to the current command line *minus* n.
!str	Refer to the most recent command containing str.
!?str[?]	Refer to the most recent command containing str.
!#	Refer to the current command line typed so far.
!{...}	Insulate a history reference form adjacent characters (if necessary).

zsh **specific**

There are several ways to use the history mechanism in zsh . One way is to use csh-style ! history operator like shown in the table 5.

If you don't want to use this you have to setopt nobanghist and you can use the arrows. Moreover, if you don't want the line in the history to be executed because you want to edit it, then you have to set the histverify option. If this option is set, then commands that result from history expansion will not be executed immediately and they will come like that:

```
zsh ~ % command-given-from-history-expansion █
```

Appendix C - `tcsh` environment variables

Some environment variables set by `tcsh` extracted from the man pages:

`HOSTTYPE`	–	symbolic name for the type of computer that the shell is running on (refer to the behaviour of the sys variable in the AFS.
`HPATH`	–	path to look for command documentation
`LANG`	–	used to give preferred character environment
`LC_CTYPE`	–	used to change only ctype character handling
`NOREBIND`	–	inhibits rebinding of printable characters to self-insert-command
`PATH`	–	path to look for command execution
`SHLVL`	–	current shell level nesting
`TERM`	–	used to tell how to handle the terminal
`LINES`	–	Number of lines in terminal
`COLUMNS`	–	Number of columns in terminal
`TERMCAP`	–	Terminal capability string
`SYSTYPE`	–	The current system type (Domain OS only)

Appendix D - tcsh shell variables

Some shell variables which are set by tcsh extracted from the man pages

addsuffix	–	add a / for directories, and a space for normal files when complete matches a name exactly. If unset don't add anything extra.
autoexpand	–	invoke the expand-history function automatically on completion.
autolist	–	list possibilities on an ambiguous completion.
autologout	–	The first word indicates number of minutes of inactivity before automatic logout. The optional second word indicates the number of minutes of inactivity after which the user will be prompted for his login password to enter a command.
fignore	–	list of file name suffixes (e.g. .o, ~) to ignore during complete.
histfile	–	If set, it contains the full path-name where a history file is read/written. It defaults to $HOME/.history. This is useful when sharing the same home directory in different machines, or if one wants to save all the histories in the tty sessions. It is usually set in .cshrc for interactive shells, because history is sourced between .cshrc and .login so that it is available from .login.
history	–	The first word of the history variable indicates the number of history events to be saved. The optional second word indicates the format history is printed (Defaults to %h\t%T\t%R\n).
listjobs	–	list all jobs when suspending. set listjobs=long, produces long format.
matchbeep	–	control beeping on completion. With matchbeep=nomatch, completion only beeps when there is no match, with matchbeep=ambiguous, beeps also when there are multiple matches, with matchbeep=notunique, beeps when there is one exact and other longer matches, with matchbeep=never, it never beeps.
nobeep	–	Disables beeping completely.
nostat	–	An list of directories that should not be stat'ed during a completion operation. Some directories e.g. "/afs", can take too much time to complete a stat(2) operation, making them prime candidates for membership in this list.
prompt	–	the string to prompt with.
recexact	–	recognize exact matches even if they are ambiguous.
rmstar	–	Prompt the user before execution of 'rm *'.
savehist	–	number of history items to save between login sessions. If it is just set, the number of items is taken from $history.
shlvl	–	Integer value indicating the number of nested shells. On login shells the level is reset to 1.
showdots	–	show hidden files in list and complete operations. If it is set to -A, then "." and ".." are not displayed, similarly to the BSD ls -A.
symlinks	–	can be set to: chase , in order to resolve the full path names, ignore , in order to use $cwd to resolve relative directory references for builtins. expand , in order to use $cwd and rewrite the arguments for commands.
term	–	the terminal type;
visiblebell	–	use the visible bell (screen flash) rather than audible bell.

`watch`	–	list of events to watch.
`who`	–	format string for the printouts generated when watch is set.

Appendix E - zsh options

Some options of zsh extracted from the manpages

append_history	–	if this is set, zsh sessions will append their history list to the history file, rather than overwrite it. thus, multiple parallel zsh sessions will all have their history lists added to the history file, in the order they are killed.
auto_cd	–	if a command is not in the hash table, and there exists an executable directory by that name, perform the cd command to that directory.
auto_list	–	automatically list choices on an ambiguous completion.
auto_menu	–	automatically use menu completion after the second consecutive request for completion, for example by pressing the *TAB* key repeatedly. this option is overridden by menu_complete.
auto_pushd	–	make cd act like pushd.
auto_resume	–	treat single word simple commands without redirection as candidates for resumption of an existing job.
cdable_vars	–	if the argument to a cd command (or an implied cd with the auto_cd option set) is not a directory, and does not begin with a slash, try to expand the expression as if it were preceded by a ~ (see filename expansion above).
chase_links	–	resolve symbolic links to their true values.
complete_aliases	–	if set, aliases on the command line are not internally substituted before completion is attempted.
complete_in_word	–	if unset, the cursor is set to the end of the word if completion is started. otherwise it stays there and completion is done from both ends.
extended_glob	–	treat the #, ~ and ^ characters as part of patterns for filename generation, etc. (an initial unquoted ~ always produces named directory expansion as in filename expansion above.)
extended_history	–	save beginning and ending timestamps to the history file. the format of these timestamps is `:<beginning time>:<ending time>:<command>`.
glob_complete	–	when the current word has a glob pattern, do not insert all the words resulting from the expansion but cycle through them like menu_complete. if no matches are found, a '*' is added to the end of the word or inserted at the cursor if complete_in_word is set, and expansion is attempted again. using patterns works not only for files but for all completions, such as options, user names, etc.
glob_dots	–	do not require a leading . in a filename to be matched explicitly.
hist_ignore_dups	–	do not enter command lines into the history list if they are duplicates of the previous event.
hist_ignore_space	–	do not enter command lines into the history list if any command on the line begins with a blank.
hist_no_store	–	remove the history (fc -l) command from the history when invoked.
ignore_eof	–	do not exit on end-of-file. require the use of exit or logout instead.
ksh_option_print	–	alters the way options settings are printed.
list_ambiguous	–	if this option is set, completions are shown only if the completions don't have a unambiguous prefix or suffix that could be inserted in the command line.

`list_types`	–	when listing files that are possible completions, show the type of each file with a trailing identifying mark.
`long_list_jobs`	–	list jobs in the long format by default.
`mail_warning`	–	print a warning message if a mail file has been accessed since the shell last checked.
`monitor`	–	allow job control. set by default in interactive shells.
`no_beep`	–	do not beep.
`no_clobber`	–	prevents `>` redirection from truncating existing files. `>!` may be used to truncate a file instead. also prevents `>>` from creating files. `>>!` may be used instead.
`no_hup`	–	don't send the hup signal to running jobs when the shell exits.
`notify`	–	report the status of background jobs immediately, rather than waiting until just before printing a prompt.
`push_ignore_dups`	–	don't push multiple copies of the same directory onto the directory stack.
`pushd_silent`	–	do not print the directory stack after pushd or popd.
`pushd_to_home`	–	have pushd with no arguments act like pushd `$HOME`.
`rc_quotes`	–	allow the character sequence `''` to signify a single quote within singly quoted strings.
`rec_exact`	–	in completion, recognize exact matches even if they are ambiguous.
`rm_star_silent`	–	do not query the user before executing "`rm *`" or "`rm path/*`".
`sh_word_split`	–	split words separated by a blank.
`sun_keyboard_hack`	–	if a line ends with a backquote, and there are an odd number of backquotes on the line, ignore the trailing backquote. this is useful on some keyboards where the return key is too small, and the backquote key lies annoyingly close to it.

Appendix F - zsh automatically set parameters

Some environment variables automatically set by zsh extrcated from the man pages:

host	–	the current hostname.
hosttype	–	a string corresponding to the type of the host the shell is running on.
shlvl	–	incremented by one each time a new shell is started.

Some shell variables automatically set by zsh

fignore (FIGNORE)	–	an array (colon separated list) containing the suffixes of files to be ignored during filename completion.
fpath	–	an array (colon separated list) of directories specifying the search path for function definitions. this path is searched when a function with the -u attribute is referenced. if an executable file is found, then it is read and executed in the current environment.
histfile	–	the file to save the history in when an interactive shell exits. if unset, the history is not saved.
histsize	–	the maximum size of the history list.
mail	–	if this parameter is set and mailpath is not set, the shell looks for mail in the specified file. by default it is set to the user's system mailbox.
mailcheck	–	the interval in seconds between checks for new mail.
mailpath (MAILPATH)	–	an array (colon-separated list) of filenames to check for new mail. each filename can be followed by a ? and a message that will be printed. the sequence $_ in the message will be replaced by the name of the mail file. the default message is "you have new mail." if an element is a directory instead of a file the shell will recursively check every file in every subdirectory of the element.
manpath (MANPATH)	–	an array (colon-separated list) whose value is not used by the shell. the manpath array can be useful, however, since setting it also sets manpath, and vice versa.
path (PATH)	–	an array (colon-separated list) of directories to search for commands. when this parameter is set, each directory is scanned and all files found are put in a hash table.
prompt	–	the primary prompt string, printed before a command is read; the default is %m%# . if the escape sequence takes an optional integer, it should appear between the '%' and the next character of the sequence.
savehist	–	the maximum number of history events to save in the history file.

Index

↑, 5, 22
**, 21
!
  history event designator, 22, 33
  history mechanism, 28
  prompt setting, 28
  redirections, 24, 26, 37
#!, 7, 16
$
  arguments, 12
  mail, 39
  quote, 24, 26
  regular expression, 25
  variable designator, 8, 10, 11, 15, 17, 28, 35, 37
  vi, 18
^
  control character, 30
  filename generation, 20
  prompt setting, 28
  quoted character, 37
  regular expression, 25
  vi, 18
^[p
  history mechanism, 22
|
  OR, 25
  pipe, 12, 24–26

alias, 12, 15, 37
arrow key, 5, 18, 22, 33

batch
  batch state, 7
bindkey, 18
  bindkey and prompt, 28
Bourne, 11, 14, 16, 31
builtin functions, 10, 12, 17, 35
  expand-glob, 20
  history, 10, 12, 21, 35
  list-glob, 20
  source, 10, 16
  function, 10

C-shell, 16, 31
command
  cat, 25
  cut, 26
  emacs, 17, 18, 27, 33
  grep, 12, 25
    egrep, 25
  man, 6, 17, 28, 34, 35, 37, 39
  rlogin, 7
  tee, 25
  telnet, 7
  vi, 18
  xterm, 7, 17, 30
completion
  compctl, 10
  fignore, 17, 35, 39
  completion mechanism, 5, 10, 17, 35, 37, 39
  environment variables completion, 17
  filename completion, 17, 35, 39
  username completion, 20
configuration files, 5, 8, 15
Ctrl A, 18
Ctrl D, 20
Ctrl E, 18
Ctrl X*, 20
Ctrl Xg, 20

environment, 5
  HOME, 15, 21, 28, 35, 37
  ZDOTDIR, 15
  environment variables, 10, 11, 15, 17, 34, 39
  variables, 11
ESC, 5
  ESC $, 27
  ESC n, 22
  ESC p, 22
  ESC s, 27
extendedglob, 37

file
  hidden files, 35
  configuration file, 5, 8, 15
  end of file, 37
  executable file, 7, 16, 17, 39
  execution of file, 16, 17
  filename completion, 17, 35, 39
  filename generation, 20, 21, 37
  history file, 21, 35, 37, 39
  list of files, 25, 37
  long filenames, 18
  mail file, 37, 39

redirections, 24–26, 37
security, 25
source of file, 16
spelling correction of filename, 27
zsh multiple redirections, 25
file descriptors, 24, 26
function keys, 18
functions, 12, 18

globbing, 20, 37

HEP, 5
history
^[p, 22
expand-history function, 17
history variable, 21, 35
history-search-backward function, 22
history-search-forward function, 22
savehist variable, 21, 35, 39
builtin history, 12
builtin function history, 10, 12, 21, 35
file .history, 35
history expansion, 33
history file, 21, 35, 37, 39
history mechanism, 5, 12, 21, 22, 33, 35, 37, 39
option histverify, 33
shell variable histfile, 21, 35, 39

initialisation files, 5, 8, 15
interactive
interactive shell, 7, 15, 17, 30, 35, 37, 39

job control
job control mechanism, 30
zsh specific, 30

login
command login, 7
command rlogin, 7
login name generation, 20
login shell, 7, 15, 21, 35
username generation, 20
logout
logout shell, 7, 8, 21, 35, 37

multiline editing
emacs mode, 18
vi mode, 18
multiline editing mechanism, 5, 18

name generation
filename generation, 20, 37
name generation mechanism, 20
username generation, 20

option, 10, 15, 21, 24, 26, 30, 33, 35, 37
tcsh specific, 14
zsh specific, 14
command option, 7

Page Down, 18, 22
Page Up, 18, 22
pipe, 12, 24, 25
PLUS, 5
prompt, 5, 17, 28, 30, 35, 37, 39

quote, 8, 12, 24, 26, 37

redirections, 24, 25, 37
!, 37
zsh, 26
zsh multiple redirections, 25

shell
environment variables, 10, 11
shell variables, 10, 17, 21, 35, 39
shell variables, 10, 17, 21, 35, 39
spelling correction, 27
correct variable or option, 27
correctall zsh option, 27
ESC $, 27
ESC s, 27
startup files, 5, 8, 15
STDERR, 24
STDIN, 24
STDOUT, 24

TAB, 5, 17
tcsh
.cshrc, 15, 35
.login, 15, 35
.logout, 15
.tcshrc, 15
csh.cshrc, 15
csh.login, 15
history, 21
logout, 15
system startup files, 15
user startup files, 15

UCO, 5

zsh

.history, 21
.zlogin, 15
.zlogout, 15
.zprofile, 15
.zshenv, 15
.zshrc, 15
zlogout, 15
zprofile, 15
zshenv, 15
zshrc, 15
environment variable ZDOTDIR, 15
option extendedglob, 21
option numericglobsort, 21
system startup files, 15
user startup files, 15