Bash Programming Pocket Reference

lazy dogs @ dogtown <dogtown@mare-system.de>

VERSION 2.2.16 :: 19 September 2012

Abstract

A quick cheat sheet for programmers who want to do shell scripting. This is not intended to teach bash-programming. based upon: http://www.linux-sxs.org/programming/bashcheat.html for beginners, see moar References at the end of this doc

Copyright Notice

(c) 2007-2012 MARE system

This manual_is free software; you may redistribute it and/or modify it under the terms of the GNU Free Documentation License as published by the Free Software Foundation; either version 1.3, or (at your option) any later version.

This is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details. A copy of the GNU Free Documentation License is available on the World Wide Web at http://www.gnu.org/licenses/fdl.txt. You can also obtain it by writing to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

GNU Free Documentation License (http://www.gnu.org/licenses/fdl.txt)

Contents

Chapter 1

Bash

1.1 Basics

All bash scripts must tell the o/s what to use as the interpreter. The first line of any script should be:

```
#!/bin/bash
```

You must either make bash scripts executable `chmod +x filename` or invoke bash with the script as argument: `bash ./your_script.sh`

1.2 Variables and getopt - get command line options

1.2.1 Variables

Create a variable - just assign value. Variables are non-datatyped (a variable can hold strings, numbers, etc. with out being defined as such). `varname=value` Display a variable via `echo` by putting $ on the front of the name; you can assign the output of a command to a variable too:

```
display:
  echo $varname

assign:
  varname=`command1 | command2 | command3`
```

Values passed in from the command line as arguments are accessed as $# where #= the index of the variable in the array of values being passed in. This array is base 1 not base 0.

```
command var1 var2 var3 .... varX
```

$1 contains whatever var1 was, $2 contains whatever var2 was, etc.

1.2.2 Built in variables:

- $1-$N :: Stores the arguments (variables) that were passed to the shell program from the command line. >
- $? :: Stores the exit value of the last command that was executed.
- $0 :: Stores the first word of the entered command (the name of the shell program).
- $* :: Stores all the arguments that were entered on the command line ($1 $2 . . .).
- "$@" :: Stores all the arguments that were entered on the command line, individually quoted (“$1” “$2” . . .).

1.2.3 getopt - command line options

```
if [ "$1" ]; then
    # options with values: o: t: i:
    # empty options: ohu
    while getopts ohuc:t:i: opt
    do
    case $opt in
    o)

        o_commands

    ;;

    u)
        u_commands
    ;;

    t)
```

```
            t_ARGS="$OPTARG"
        ;;

        *)
            exit
        ;;


        esac
        done

    fi

    shift $((OPTIND - 1))
```

1.3 Quote Marks

Regular double quotes "like these" make the shell ignore whitespace and count it all as one argument being passed or string to use. Special characters inside are still noticed/obeyed.

Single quotes 'like this' make the interpreting shell ignore all special characters in whatever string is being passed. The back single quote marks (aka backticks) (`command`) perform a different function. They are used when you want to use the results of a command in another command. For example, if you wanted to set the value of the variable contents equal to the list of files in the current directory, you would type the following command: `contents=`ls``, the results of the ls program are put in the variable contents.

1.4 Tests / Comparisons

A command called test is used to evaluate conditional expressions, such as a if-then statement that checks the entrance/exit criteria for a loop.

```
test expression

  ... or ...

[ expression ]
```

```
USAGE:

[ expression ] && do_commands
        => do_commands if expresssion is ok
```

1.4.1 Numeric Comparisons

```
int1 -eq int2   Returns True if int1 is equal to int2.
int1 -ge int2   Returns True if int1 is greater than or equal to int2.
int1 -gt int2   Returns True if int1 is greater than int2.
int1 -le int2   Returns True if int1 is less than or equal to int2
int1 -lt int2   Returns True if int1 is less than int2
int1 -ne int2   Returns True if int1 is not equal to int2
```

1.4.2 String Comparisons

```
str1 = str2     Returns True if str1 is identical to str2.
str1 != str2    Returns True if str1 is not identical to str2.
str             Returns True if str is not null.
-n str          Returns True if the length of str is
                greater than zero.
-z str          Returns True if the length of str is equal
                to zero. (zero is different than null)
```

1.4.3 File Comparisons

```
-d filename     Returns True if filename is a directory.
-e filename     Returns True if filename exists (might be a directory
-f filename     Returns True if filename is an ordinary file.
-h filename     Returns True if filename is a symbolic link
-p filename     Returns True if filename is a pipe
-r filename     Returns True if filename can be read by the process.
-s filename     Returns True if filename has a nonzero length.
-S filename     Returns True if filename is a Socket
-w filename     Returns True if file, filename can be written by the process.
```

```
-x filename     Returns True if file, filename is executable.

$fd1 -nt $fd2   Test if fd1 is newer than fd2. The modification date is used
$fd1 -ot $fd2   Test if fd1 is older than fd2. The modification date is used
$fd1 -ef $fd2   Test if fd1 is a hard link to fd2
```

1.4.4 Expression Comparisons

```
!expression     Returns true if expression is not true
expr1 -a expr2  Returns True if expr1 and expr2 are true.
                ( && , and )
expr1 -o expr2  Returns True if expr1 or expr2 is true.
                ( ||, or )
```

1.4.5 testing if $var is an integer

src1: http://www.linuxquestions.org/questions/programming-9/test-for-integer-in-bash-279227/#post1514631

src2: http://stackoverflow.com/questions/806906/how-do-i-test-if-a-variable-is-a-number-in-bash

```
You can also use expr to ensure a variable is numeric

a=100

if [ `expr $a + 1 2> /dev/null` ] ; then
    echo $a is numeric ;
else
    echo $a is not numeric ;
fi


example 2:

[[ $1 =~ "^[0-9]+$" ]] && echo "number" && exit 0 || echo "not a number" && exit
```

1.5 Logic and Loops

1.5.1 if ... then ... elif ... else

```
-> you can always write: (( if [ expression ]; then )) as shortcut

if [ expression ]
    then
    commands
fi

if [ expression ]
    then
    commands
else
    commands
fi

if [ expression ]
    then
    commands
elif [ expression2 ]
    then
    commands
else
    commands
fi


# arithmethic in if/test

function mess {
   if (( "$1" > 0 )) ; then
      total=$1
   else
      total=100
   fi
   tail -$total /var/log/messages | less

}
```

1.5.2 Loops

it can be usefull to assign IFS_Values for your script before running and reassign default-values at the end.

```
ORIGIFS=$IFS
IFS=`echo -en " \n\b"`

for var1 in list
    do
    commands
done
IFS=$ORIGIFS
```

This executes once for each item in the list. This list can be a variable that contains several words separated by spaces (such as output from ls or cat), or it can be a list of values that is typed directly into the statement. Each time through the loop, the variable var1 is assigned the current item in the list, until the last one is reached.

```
while [ expression ]
    do
    commands
done

until [ expression ]
    do
    commands
done
```

1.5.3 Case select

```
case string1 in
    str1)
        commands1
        ;;
    str2)
```

```
        commands2
        ;;
    *)
        commands3
        ;;
esac
```

string1 is compared to str1 and str2. If one of these strings matches string1, the commands up until the double semicolon (; ;) are executed. If neither str1 nor str2 matches string1, the commands associated with the asterisk are executed. This is the default case condition because the asterisk matches all strings.

1.5.4 select -> select from a list of values

```
export PS3="
 alternate_select_prpmpt # > "
"
select article_file in $sgml_files
    do
    case $REPLY in
    x)
    exit
    ;;
    q)
    exit
    ;;
    esac
    NAME="$article_file"
    break
    done



fi
```

1.6 bash foo

1.6.1 input/output-redirection

Three le descriptors (0, 1 and 2) are automatically opened when a shell in invoked. They represent:

```
 0    standard input (stdin)
1    standard output (stdout)
2    standard error (stderr)
```

A commandâs input and output may be redirected using the following notation:

```
 <file    take input from le
>file     write output to le
          (truncate to zero if it exists)
>>file    append output to le, else create
<<word    \here" document; read input until line matches word
<>file    open le for reading and writing
<&digit   use le descriptor digit as input
          (>&digit for output)
<&-       close standard input (>&- close output)
cmd1|cmd2 stdout of cmd1 is piped to stdin of cmd2
```

```
 ls -l >listing
```

```
ls -l | lpr
zcat file.tar.Z | tar tvf -
```

1.6.2 Functions

Create a function:

```
fname(){
    commands
}
```

you can call then the function `fname`, giving $ARGS as $1 $2

1.6.3 read user input

In many ocations you may want to prompt the user for some input, and there are several ways to achive this. This is one of those ways. As a variant, you can get multiple values with read, the second example may clarify this.

```
#!/bin/bash
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
```

1.6.4 reading return values / outputs from commands

In bash, the return value of a program is stored in a special variable called $?. This illustrates how to capture the return value of a program, I assume that the directory dada does not exist. (This was also suggested by mike)

```
#!/bin/bash
cd /dada &> /dev/null
echo rv: $?
cd $(pwd) &> /dev/null
echo rv: $?
```

Capturing a commands output

This little scripts show all tables from all databases (assuming you got MySQL installed).

```
#!/bin/bash
DBS=`mysql -uroot  -e"show databases"`
for b in $DBS ;
do
        mysql -uroot -e"show tables from $b"
done
```

1.6.5 Arithmetic & Bash-Expansion

```
i=$(( i + 1 ))
let i+=1
i=$(( i++))
let i++


Operators:

Op  Operation with assignment   Use     Meaning
-----------------------------------------------
=   Simple assignment           a=b     a=b
*=  Multiplication              a*=b    a=(a*b)
/=  Division                    a/=b    a=(a/b)
%=  Remainder                   a%=b    a=(a%b)
+=  Addition                    a+=b    a=(a+b)
-=  Subtraction                 a-=b    a=(a-b)
```

1.6.6 using Arrays

src: http://www.softpanorama.org/Scripting/Shellorama/arithmetic_expressions.shtml

Initialization of arrays in bash has format similar to Perl:

```
solaris=(serv01 serv02 serv07 ns1 ns2)
```

Each element of the array is a separate word in the list enclosed in parentheses. Then you can refer to each this way:

```
echo solaris is installed on ${solaris[2]}
```

If you omit index writing echo $solaris you will get the first element too. Another example taken from Bash Shell Programming in Linux

```
array=(red green blue yellow magenta)

len=${#array[*]}
echo "The array has $len members. They are:"
```

```
i=0
while [ $i -lt $len ]; do
    echo "$i: ${array[$i]}"
    let i++
done
```

1.6.7 date & time - conversion

```
get date in iso-formate
    now_time=`date +%F - %H:%M:%S`

get unix_timestamp
    unix_time=`date %s`

convert unix-timestamp to iso-date
    date --date "1970-01-01 $unix_time sec" "+%Y-%m-%d %T"

date_strftime - macros / format-controls:
   %%     a literal %
   %a     localeâs abbreviated weekday name (e.g., Sun)
   %A     localeâs full weekday name (e.g., Sunday)
   %b     localeâs abbreviated month name (e.g., Jan)
   %B     localeâs full month name (e.g., January)
   %c     localeâs date and time (e.g., Thu Mar  3 23:05:25 2005)
   %C     century; like %Y, except omit last two digits (e.g., 21)
   %d     day of month (e.g, 01)
   %D     date; same as %m/%d/%y
   %e     day of month, space padded; same as %_d
   %F     full date; same as %Y-%m-%d
   %g     last two digits of year of ISO week number (see %G)
   %G     year of ISO week number (see %V); normally useful only with %V
   %h     same as %b
   %H     hour (00..23)
   %I     hour (01..12)
   %j     day of year (001..366)
   %k     hour ( 0..23)
   %l     hour ( 1..12)
   %m     month (01..12)
   %M     minute (00..59)
   %n     a newline
```

```
    %N     nanoseconds (000000000..999999999)
    %p     localeâs equivalent of either AM or PM; blank if not known
    %P     like %p, but lower case
    %r     localeâs 12-hour clock time (e.g., 11:11:04 PM)
    %R     24-hour hour and minute; same as %H:%M
    %s     seconds since 1970-01-01 00:00:00 UTC
    %S     second (00..60)
    %t     a tab
    %u     day of week (1..7); 1 is Monday
    %U     week number of year, with Sunday as first day of week (00..53)
    %V     ISO week number, with Monday as first day of week (01..53)
    %w     day of week (0..6); 0 is Sunday
    %W     week number of year, with Monday as first day of week (00..53)
    %x     localeâs date representation (e.g., 12/31/99)
    %X     localeâs time representation (e.g., 23:13:48)
    %y     last two digits of year (00..99)
    %Y     year
    %z     +hhmm numeric timezone (e.g., -0400)
    %:z    +hh:mm numeric timezone (e.g., -04:00)
    %::z   +hh:mm:ss numeric time zone (e.g., -04:00:00)
    %:::z  numeric time zone with : to necessary precision (e.g., -04, +05:30)
    %Z     alphabetic time zone abbreviation (e.g., EDT)
```

1.6.8 parsing a simple conf in bash

src: http://www.chimeric.de/blog/2007/1122_parsing_simple_config_files_in_bash

The function uses some of the more advanced bash features like parameter substitution a.s.o. which I won't explain here. For a good read on the whole bash scripting topic I recommend the Advanced Bash Scripting Guide.

```
# simple configuration file
#
# default settings
default {
    DATE_PREFIX=$(date -I)
    EXT_FULL="full"
    EXT_DIFF="diff"
    SSHFS_OPTS="-C"
    DAR_OPTS="-v -m 256 -y -s 600M -D"
    DAR_NOCOMPR="-Z '*.gz' -Z '*.bz2' -Z '*.zip' -Z '*.png'"
}
```

```
# backup target system
system {
    SRC_DIR="/"
    DEST_DIR="/mnt/data/backups/tatooine"
    PREFIX="system"
    TYPE="R"
    HOST="chi$@coruscant"
}

# backup target home
home {
    SRC_DIR="/home/user"
    DEST_DIR="/mnt/data/backups/tatooine"
    PREFIX="home-nomedia"
    TYPE="R"
    HOST="chi$@coruscant"
    DAR_EXCLUDES="media"
}

------------------

#!/usr/bin/env bash
# $@author Michael Klier chi$@chimeric.de

function readconf() {

    match=0

    while read line; do
        # skip comments
        [[ ${line:0:1} == "#" ]] && continue

        # skip empty lines
        [[ -z "$line" ]] && continue

        # still no match? lets check again
        if [ $match == 0 ]; then

            # do we have an opening tag ?
            if [[ ${line:$((${#line}-1))} == "{" ]]; then
```

```
                # strip "{"
                group=${line:0:$((${#line}-1))}
                # strip whitespace
                group=${group// /}

                # do we have a match ?
                if [[ "$group" == "$1" ]]; then
                    match=1
                    continue
                fi

                continue
            fi

        # found closing tag after config was read - exit loop
        elif [[ ${line:0} == "}" && $match == 1 ]]; then
            break

        # got a config line eval it
        else
            eval $line
        fi

    done < "$CONFIG"
}

CONFIG="/home/user/.sampleconfig"

readconf "default"

echo $DATE_PREFIX
echo $DAR_OPTS
echo $DAR_NOCOMPR
```

1.6.9 extracting filenames from path/urls:

```
url="http://www.emergingthreats.org/rules/emerging_all.rules"
rules_name=${url##*/}
# $rules_name _> emerging_all.rules
wget -O $rules_name $url
```

```
path="/var/log/some.log"
file_name=${path##*/}
```

1.6.10 extracting/deleting first/latest char from string:

src: http://blog.pregos.info/2011/10/06/bash-delete-last-character-from-string/

```
Print last char from string:

    user@desktop:~$ VAR=foobar
    user@desktop:~$ echo $VAR
    foobar
    user@desktop:~$ echo ${VAR: -1}
    r


Delete last character from string:

    user@desktop:~$ VAR=foobar
    user@desktop:~$ echo $VAR
    foobar
    user@desktop:~$ echo ${VAR%?}
    fooba


Delete first character from string

    user@desktop:~$ VAR=foobar
    user@desktop:~$ echo $VAR
    foobar
    user@desktop:~$ echo ${VAR:1}
    oobar
```

1.7 usefull Shell-Commands

stuff like awk, sed etc

1.7.1 crontab - adds from commandline

src: http://dbaspot.com/solaris/386215-adding-line-crontab-command-line.html

```
Re: Adding line in crontab from command line...

On Wed, 9 Apr 2008 11:17:47 -0700 (PDT), contracer11@gmail.com wrote:
>
> Hi:
>
> Can you tell me if is there any way to make this task ?
>
>
> 00 1 * * * /monitor_file_system 2>/dev/null > crontab
>

crontab -l | (cat;echo "00 1 * * * /monitor_file_system") | crontab

Helmut

--
Almost everything in life is easier to get into than out of.
(Agnes' Law)
```

1.7.2 sed-examples

src: http://www.grymoire.com/Unix/Sed.html

SED is a tool to manipulate text-streams (Stream EDitor), together with redirects it might be used to substitute text from/ to files.

The character after the s is the delimiter. It is conventionally a slash, because this is what ed, more, and vi use. It can be anything you want, however. If you want to change a pathname that contains a slash - say /usr/local/bin to /common/bin - you could use the backslash to quote the slash:

```
$ sed 's[delimiter]ot[delimiter]nt[delimiter]{flags} < input > output

$ sed 's/\/usr\/local\/bin/\/common\/bin/' < old >new
$ sed 's_/usr/local/bin_/common/bin_' < old >new
```

```
 $ sed 's:/usr/local/bin:/common/bin:' < old >new
 $ sed 's|/usr/local/bin|/common/bin|' < old >new

combining commands:

sed -e 's/a/A/' -e 's/b/B/' < old >new

seing multiples files (f1..3)
 $ sed 's/^#.*//' f1 f2 f3 | grep -v '^$' | wc -l

grep-simulation: Nothing is printed, except those lines with PATTERN included.
 $ sed -n 's/PATTERN/&/p' file

The simplest restriction is a line number. If you wanted to delete the
first number on line 3, just add a "3" before the command:

 $ sed '3 s/[0-9][0-9]*//' < file >new

restrict to the first 100 lines:

 $ sed '1,100 s/A/a/'

pexecute from line 101 until the end; "$"  means the last
line in the file.

 $ sed '101,$ s/A/a/'
```

Pick one you like. As long as it's not in the string you are looking for, anything goes. And remember that you need three delimiters. If you get a "Unterminated 's' command" it's because you are missing one of them.

SED-FLAGS

with no flags given the fiorst found pattern is changed.

```
    /g    -> global sustitution (every occurance) instead of the first one
    /2    -> change only the second pattern
    /2g   -> change everythiong from the 2nd pattern onwards
    /p    -> print foudn matches (sed -n .../p -> simulate grep
    /w fd -> write outpuf to file $fd
```

Chapter 2

Regular Expressions

2.1 POSIX Character Classes for Regular Expressions & their meanings

Class	Meaning
[:alpha:]	Any letter, [A-Za-z]
[:upper:]	Any uppercase letter, [A-Z]
[:lower:]	Any lowercase letter, [a-z]
[:digit:]	Any digit, [0-9]
[:alnum:]	Any alphanumeric character, [A-Za-z0-9]
[:xdigit:]	Any hexadecimal digit, [0-9A-Fa-f]
[:space:]	A tab, new line, vertical tab, form feed, carriage return, or space
[:blank:]	A space or a tab.
[:print:]	Any printable character
[:punct:]	Any punctuation character: ! ’ # S % & ’ ( ) * + , − . / : ; < = > ? @ [/] ^ _ { | } ~
[:graph:]	Any character defined as a printable character except those defined as part of the space character class
[:word:]	Continuous string of alphanumeric characters and underscores.
[:ascii:]	ASCII characters, in the range: 0-127
[:cntrl:]	Any character not part of the character classes: [:upper:], [:lower:], [:alpha:], [:digit:], [:punct:], [:graph:], [:print:], [:xdigit:]

2.2 Special Characters in Regular Expressions

```
Char      Meaning                                        Example
*         Match zero, one or more of the previous          1
?         Match zero or one of the previous                2
+         Match one or more of the previous                3
\         Used to escape a special character               4
.         Wildcard character, matches any character        5
( )       Group characters                                 6
[ ]       Matches a range of characters                    7
[0-9]+    matches any positive integer
[a-zA-Z]  matches ascii letters a-z (uppercase and lower case)
[^0-9]    matches any character not 0-9.
|         Matche previous OR next character/group          8
{ }       Matches a specified number of occurrences        9
^         Beginning of a string / set                     10
$         End of a string.                                11

Examples:
    1  -  Ah* matches "Ahhhhh" or "A"
    2  -  Ah? matches "Al" or "Ah"
    3  -  Ah+ matches "Ah" or "Ahhh" but not "A"
    4  -  Hungry\? matches "Hungry?"
    5  -  do.* matches "dog", "door", "dot", etc.
    6  -  see 8
    7  -  [cbf]ar matches "car", "bar", or "far"
    8  -  (Mon)|(Tues)day matches "Monday" or "Tuesday"
    9  -  [0-9]{3} matches "315" but not "31"
          [0-9]{2,4} matches "12", "123", and "1234"
          [0-9]{2,} matches "1234567..."
   10  -  ^http matches strings that begin with http, such as a url.
          [^0-9]    matches any character not 0-9.
   11  -  ing$ matches "exciting" but not "ingenious"
```

2.3 Usefulle RegExes

- Email: [_a-zA-Z0-9-]+(\.[_a-zA-Z0-9._-]+)*@([a-zA-Z0-9._-]+\.)+([a-zA-Z]{2,4})
- IP: /^(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})$/

Chapter 3

Editor - Quick References

3.1 Emacs Refernces

3.1.1 Basics

- **C-x-f** find file (open file)
- **C-x-s** save current buffer
- **C-x s** save buffers that have been altered
- **C-x-w** save as
- **C-x-c** quit

3.1.2 Help

- **C-h a cmd** Get help on command
- **C-h k** Answers 'what does this key combination do?'

3.1.3 Killing and yanking

- **C-k** kill to end of line
- **M-d** kill to end of word (adding to kill ring)
- **M-delete** back-kill to start of word (adding to kill ring)

- **C-y** yank (paste)
- **M-y** yank previous(Do C-y M-y M-y M-y to yank third last kill)
- **C-space** start mark
- **M-w** kill from mark to here
- **C-insert** copy as kill from mark to here

3.1.4 Navigating

- **C-e** goto end of current line
- **C-a** goto beginning of current line
- **M-<** goto beginning buffer
- **M->** goto end of buffer
- **M-x** goto-line RET goes to specified line
- **C-M-f** goto closing brace (standing on the opening brace)
- **C-M-b** goto opening brace (standing on the closing brace)
- **C-u C-space** which takes you back to wherever you were previously working
- **M-m** position cursor on start of indentation

3.1.5 Window/Buffer commands

- **C-x 2** Split window horizontally
- **C-x 3** Split window vertically
- **C-x 1** Close all other windows but the one where the cursor is
- **C-x 0** (Zero)Close this window, keep the other
- **C-x o** (oh!) Jump to next window
- **C-x b** View another buffer
- **C-x-b** Pick another buffer to view
- **C-l** Center buffer around line
- **C-u 1 C-l** Make this line the top line of the buffer

3.1.6 Search and replace

- **C-s** incremental search forward
- **C-r** incremental search backward
- **M-%** Query replace
- **C-M-%** Query replace regexp
- **M-x** occur Find regexp in current buffer
- **M-x** grep-find Find regexp recursively from a directory
- **M-x** occur find occurences in this file, present as a list

3.1.7 Miscellaneous

- **C-g** quit whatever command (you did something you did not intend)
- **M-** remove white space before and after cursor
- **M-^** join this line with the prevoius and fix white space
- **M-x** delete-trailing-whitespace removes blanks after last char on all lines
- **C-x C-t** transpose lines, move the current line one line upwards
- **M-t** transpose words, swap the word behind the cursor for the one after
- **M-l** make the rest of this word lower case
- **M-u** make the rest of this word lower case
- **C-x C-l** make region lower case
- **C-x C-u** make region upper case

3.1.8 Navigating code

- **M-.** Jump to tag (where does this function reside? go there!)
- **M-x** goto-line RET goes to specified line

3.2 vi pocket Reference

src: http://www.lagmonster.org/docs/vi.html

command_mode: [ESC]

3.2.1 Modes

Vi has two modes insertion mode and command mode. The editor begins in command mode, where the cursor movement and text deletion and pasting occur. Insertion mode begins upon entering an insertion or change command. [ESC] returns t he editor to command mode (where you can quit, for example by typing :q!). Most commands execute as soon as you type them except for "colon" commands which execute when you press the ruturn key.

3.2.2 File Handling

- **:w** Write file
- **:w!** Write file (ignoring warnings)
- **:w!** file Overwrite file (ignoring warnings)
- **:wq** Write file and quit
- **:q** Quit
- **:q!** Quit (even if changes not saved)
- **:w file** Write file as file, leaving original untouched
- **ZZ** Quit, only writing file if changed
- **:x** Quit, only writing file if changed
- **:n1,n2w file** Write lines n1 to n2 to file
- **:n1,n2w » file** Append lines n1 to n2 to file
- **:e file2 Edit file2** (current file becomes alternate file)
- **:e!** Reload file from disk (revert to previous saved version)
- **:e#** Edit alternate file
- **%** Display current filename

- **#** Display alternate filename
- **:n** Edit next file
- **:n!** Edit next file (ignoring warnings)
- **:n files** Specify new list of files
- **:r file** Insert file after cursor
- **:r !command** Run command, and insert output after current line

3.2.3 Quitting

- **:x** Exit, saving changes
- **:q** Exit as long as there have been no changes
- **ZZ** Exit and save changes if any have been made
- **:q!** Exit and ignore any changes

3.2.4 Inserting Text

- **i** Insert before cursor
- **I** Insert before line
- **a** Append after cursor
- **A** Append after line
- **o** Open a new line after current line
- **O** Open a new line before current line
- **r** Replace one character
- **R** Replace many characters

3.2.5 Motion

- **h** Move left
- **j** Move down
- **k** Move up
- **l** Move right
- **w** Move to next word
- **W** Move to next blank delimited word
- **b** Move to the beginning of the word
- **B** Move to the beginning of blank delimted word
- **e** Move to the end of the word
- **E** Move to the end of Blank delimited word
- **(** Move a sentance back
- **)** Move a sentance forward
- { Move a paragraph back
- } Move a paragraph forward
- **0** Move to the begining of the line
- **$** Move to the end of the line
- **1G** Move to the first line of the file
- **G** Move to the last line of the file
- **nG** Move to nth line of the file
- **:n** Move to nth line of the file
- **fc** Move forward to c
- **Fc** Move back to c
- **H** Move to top of screen
- **M** Move to middle of screen
- **L** Move to botton of screen
- **%** Move to associated ( ), { }, [ ]

3.2.6 Deleting Text

Almost all deletion commands are performed by typing d followed by a motion. For example, dw deletes a word. A few other deletes are:

- **x** Delete character to the right of cursor
- **X** Delete character to the left of cursor
- **D** Delete to the end of the line
- **dd** Delete current line
- **:d** Delete current line

3.2.7 Yanking Text

Like deletion, almost all yank commands are performed by typing y followed by a motion. For example, y$ yanks to the end of the line. Two other yank commands are:

- **yy** Yank the current line
- **:y** Yank the current line

3.2.8 Buffers

Named buffers may be specified before any deletion, change, yank or put command. The general prefix has the form “c where c is any lowercase character. for example, ”adw deletes a word into buffer a. It may thereafter be put back into text with an appropriate “ap.

3.2.9 Search for strings

- **/string** Search forward for string
- **?string** Search back for string
- **n** Search for next instance of string
- **N** Search for previous instance of string

3.2.10 Replace

The search and replace function is accomplished with the :s command. It is commonly used in combination with ranges or the :g command (below).

- **:s/pattern/string/flags** Replace pattern with string according to flags.
- **g Flag** - Replace all occurences of pattern
- **c Flag** - Confirm replaces.
- **&** Repeat last :s command

3.2.11 Regular Expressions

- **. (dot)** Any single character except newline
- * zero or more occurances of any character
- **[...]** Any single character specified in the set
- **[^...]** Any single character not specified in the set
- **^** Anchor - beginning of the line
- **$** Anchor - end of line
- **\<** Anchor - begining of word
- **\>** Anchor - end of word
- **\(...\)** Grouping - usually used to group conditions
- **\n** Contents of nth grouping
- ——————————————————————————
- **[...]** Set Examples [A-Z] The SET from Capital A to Capital Z
- **[a-z]** The SET from lowercase a to lowercase z
- **[0-9]** The SET from 0 to 9 (All numerals)
- **[./=+]** The SET containing . (dot), / (slash), =, and +
- **[-A-F]** The SET from Capital A to Capital F and the dash (dashes must be specified first)
- **[0-9 A-Z]** The SET containing all capital letters and digits and a space
- **[A-Z][a-zA-Z]** In the first position, the SET from Capital A to Capital Z In the second character position, the SET containing all letters

3.2.12 Regular Expression Examples

- **/Hello/** Matches if the line contains the value Hello
- **/^TEST$/** Matches if the line contains TEST by itself
- **/^[a-zA-Z]/** Matches if the line starts with any letter
- **/^[a-z].*/** Matches if the first character of the line is a-z and there is at least one more of any character following it
- **/2134$/** Matches if line ends with 2134
- **/\(21|35\)/** Matches is the line contains 21 or 35; Note the use of ( ) with the pipe symbol to specify the 'or' condition
- **/[0-9]*/** Matches if there are zero or more numbers in the line
- **/^[^#]/** Matches if the first character is not a # in the line

Notes:

1 Regular expressions are case sensitive

2 Regular expressions are to be used where pattern is specified

3.2.13 Counts

Nearly every command may be preceded by a number that specifies how many times it is to be performed. For example, 5dw will delete 5 words and 3fe will move the cursor forward to the 3rd occurence of the letter e. Even insertions may be repeated conveniently with thismethod, say to insert the same line 100 times.

3.2.14 Other

```
~       Toggle upp and lower case
J       Join lines
.       Repeat last text-changing command
u       Undo last change
U       Undo all changes to line
```

Chapter 4

Links and Resources

4.1 Links and Resources

- Advanced Bash Scripting guide: http://tldp.org/guides.html#abs