Bash Style Guide and Coding Standard

With complementing comments on the testing of scripts

Fritz Mehner
Fachhochschule Südwestfalen, Iserlohn
mehner@fh-swf.de

Contents

Often, script programming is considered to generate only rapid and hardly to understand "throw-away solutions" which do not have to meet any specific quality requirements. However it is often ignored that there are many areas where long-living scripts are quite the rule: System management, start and configuration of operating systems, software installations, automation of user tasks and so on. And, of course, these solutions need to be maintained, extended and documented.
When it comes to script programming therefore the same requirements have to be fulfilled than when programming in a production language (meeting the purpose, correctness of the solution, meeting the requirements, robustness, maintainability) and therefore the same standards have to be employed. A program can only be maintained, if its structure and effect can be understood easily by someone who has not written the program, so that successful changes can be executed in reasonable time. The way these requirements are met depends to a great part on the programming style used. The specifications dealt with here serve primarily for generating easily understandable and thus maintainable code and therefore must be observed.

1 Length of line

The total length of a line (including comment) must not exceed more than **88 characters**. Thus searching in cross direction can be avoided and the file can be printed with the usual width of paper without lines being cut or line breaks. Instructions have to be split up, as applicable, texts can be made up as well.

2 Indentation

The indentation of program constructions has to agree with the logic nesting depth. The indentation of one step usually is in line with the tabulator steps of the editor selected. In most cases 2, 4 or 8 are chosen.

3 Comments

3.1 Introductory comments in files

Every file must be documented with an introductory comment that provides information on the file name and its contents:

```
#!/bin/bash
#===============================================================================
#
#         FILE:  stale-links.sh
#
#        USAGE:  stale-links.sh [-d] [-l] [-oD logfile] [-h] [starting directories]
#
#  DESCRIPTION:  List and/or delete all stale links in directory trees.
#                The default starting directory is the current directory.
#                Don't descend directories on other filesystems.
#
#      OPTIONS:  see function 'usage' below
# REQUIREMENTS:  ---
#         BUGS:  ---
#        NOTES:  ---
#       AUTHOR:  Dr.-Ing. Fritz Mehner (Mn), mehner@fh-swf.de
#      COMPANY:  FH Südwestfalen, Iserlohn
#      VERSION:  1.3
#      CREATED:  12.05.2002 - 12:36:50
#     REVISION:  20.09.2004
#===============================================================================
```

If required, additional information has to be added (e.g. copyright note, project assignment).

3.2 Line end comments

Consecutive line end comments start in the same column. A blank will always follow the introductory character of the comment `#` to simplify the detection of the beginning of the word.

```
found=0                         # count links found
deleted=0                       # count links deleted
```

3.3 Section comments

If several lines form a section with interlinked instructions, such section must be provided with a section comment.

```
#------------------------------------------------------------------------
#  delete links, if demanded write logfile
#------------------------------------------------------------------------
if [ "$action" == 'd' ] ; then
  rm --force "$file" && ((deleted++))
  echo "removed link :  '$file'"
  [ "$logfile" != "" ] &&   echo  "$file" >> "$logfile"
fi
```

The depth of the indentation of the comment is in accordance with the nesting depth of the subsequent instruction.

3.4 Function comments

Each function is described by an introductory comment. This comment contains the function name, a short description and the description of the parameters (if available). The name of the author and the date of issue should be added in case of subsequent amendments.

```
#===  FUNCTION  ================================================================
#         NAME:  usage
#  DESCRIPTION:  Display usage information for this script.
# PARAMETER  1:  ---
#===============================================================================
```

3.5 Commenting style

For the scope and style of the comments the following applies:

> **Short, concise and sufficiently accurate.**

Comprehensive descriptions are a matter of external documentation. The structure or the trick used is described only in exceptional cases. For instructions the following applies:

> **The comment describes the purpose of the instruction.**

The structure or the trick used is described only in exceptional cases. The following comment is not particularly helpful as it repeats only what is indicated at the beginning of the line:

```
[ "$logfile" != "" ] && $(> "$logfile")         # variable $logfile empty ?
```

The comment below, however, states the intention concisely:

```
[ "$logfile" != "" ] && $(> "$logfile")         # empty an existing logfile
```

4 Variables and constants

4.1 Use of variables

For variables meaningful, self-documenting names have to be used (such as `inputfile`). In names the first 31 characters have to be different. Long names are structured by underscores to improve legibility.
If a name is not self-documenting, the meaning and use must be described when it first occurs by a comment.

4.2 Use of constants

Principally, the following applies for all programming languages: **No constants must be included in the program text!** In particular numeral constants do not have another immediate meaning apart from their value. The meaning of the value will only become clear in the specific text context. In case of value changes of multiple occurring constants an automatic replacement in the editor is not possible, because the value might have been used in different meanings. Such program texts therefore are difficult to maintain. For the handling of constants - and of course also constant texts (such as file names) - the following recommendations apply:

Global constants and texts. Global constants and texts (e.g. file names, replacement values for call parameters and the like) are collected in a separate section at the beginning of the script and commented individually, if the number is not too high.

```
startdirs=${@:-.}               # default start directory: current directory
action=${action:-l}             # default action is list
```

Long continuous texts. Long continuous texts (e.g. descriptions, help for invocation options) can be described as **here** documents.

```
  cat <<- EOT
  List and/or delete all stale links in directory trees.

  usage : $0 [-d] [-oD logfile] [-l] [-h] [starting directories]

   -d    delete stale links
   -l    list stale links (default)
   -o    write stale links found into logfile
   -D    delete stale links listed in logfile
   -h    display this message
EOT
```

5 Success verification

5.1 Command line options

If there has to be a minimum number or a specified number of command line parameters, this number has to be verified. In case of an error, the script will be terminated with an error message and/or an indication of the required call parameters.
The parameter values handed over have to be checked for validity as well. If, for example, the name of an entry file is handed over, it has to be checked before read access whether this file exists and can be read (e.g. with the test `[ -r $inputfile ]`).

5.2 Variables, commands and functions

Variables must have obtained a meaningful starting value before being used. This has to be checked:

```
[ -e "$1" ] && expand --tabs=$number "$1" > "$1.expand"
```

This line checks whether the file exists whose name has been handed over in parameter `$1`. Since the evaluation of logic terms is aborted, as soon as the result has been determined (the "short circuit evaluation") further processing is omitted if the prior condition is false. The return value of the last command is stored in the variable `$?` and can be used for further process control:

```
mkdir "$new_directory"  2> /dev/null
if [ $? -ne 0 ]
then
    ...
fi
```

If it has not been possible to create a directory in this example, the return value of `mkdir` is not equal to zero. The variable `$?` is used also to check the return value of a function.

5.3 Execution and summary reports

Scripts which will be used interactively should display a summary report. It tells whether the script ran properly and it can be used to check the plausibility of the results, e.g.

```
mn4:~/bin # ./stale-links -o stale-links.log /opt

    ... searching stale links ...
1. stale link:  '/opt/dir link 23'
2. stale link:  '/opt/file link 71'
3. stale link:  '/opt/file link 7'

      stale links   found : 3
      stale links deleted : 0
      logfile: 'stale-links.log'
```

Detailed execution reports are written into logfiles. They need also to be available to help diagnose failure.

6 Files

File names File names: For file names meaningful basic names have to be chosen. The file extensions should, if possible, indicate the contents (`.dat`, `.log`, `.lst`, `.tmp` etc.).

Temporary files Temporary files for storing of comprehensive intermediate results are usually generated in the central directory `tmp` and removed again after their use. For generation of accidental names `mktemp` can be used (see `man 1 mktemp`):

```
#------------------------------------------------------------------------------
#  Cleanup temporary file in case of keyboard interrupt or termination signal.
#------------------------------------------------------------------------------
function cleanup_temp {
  [ -e $tmpfile ] && rm --force $tmpfile
  exit 0
}

trap  cleanup_temp  SIGHUP SIGINT SIGPIPE SIGTERM

tmpfile=$(mktemp) || { echo "$0: creation of temporary file failed!"; exit 1; }

# ... use tmpfile ...

rm --force $tmpfile
```

The function `cleanup_temp` will be called if one of the specified signals is caught from the **`trap`** statement. This function then deletes the temporary file. The file survives a termination with `SIGKILL` because this signal can not be caught.

Backup copies If several older copies of files have to be retained, the use of the date as a component of the file names of the copies is recommended:

```
timestamp=$(date +"%Y%m%d-%M%S")              # generate timestamp : YYYYMMDD-mmss

mv logfile logfile.$timestamp
```

The file `logfile` is now being renamed, e.g. in `logfile.20041210-3116`. The components of date and time are organized in reversed order. The files named following this convention are listed in directory lists sorted chronologically.

Intermediate results Intermediate results that are also written into the files can be output by the use of `tee` as well as to the default output. In this way they can serve for process control or for testing of the script:

```
echo $output_string | tee --append  $TMPFILE
```

7 Command line options

Invoking of external programs Invoking of external programs: If system programs are invoked, **the long forms of command line options** have to be used in a script, if these are available. The long names are mostly self-documenting and thus simplify reading and understanding a script. In the following `useradd`-instruction the long forms of the commands `-c`, `-p` and `-m` are used:

```
useradd --comment      $full_name           \
        --password     $encrypted_password  \
        --create-home                       \
        $loginname
```

By means of carriage return (character `\` at line end) the generation of an overlong line is avoided. The table orientation increases legibility.

Command line options of own script For the designation of own options (short form) letters that come to mind easily or that are commonly used must be selected (e.g. `-f`, with argument, for the indication of a file, or `-d`, with or without argument, for control of the extent of test outputs (debug)). For suggestions for long forms, see the **GNU Coding Standards**[1].

8 Use of Shell Builtin Commands

If possible shell buitins should be preferred to external utilities. Each call of `sed`, `awk`, `cut` etc. generates a new process. Used in a loop this can extend the execution time considerably. In the following example the shell parameter expansion is used to get the base name and the directory of a path:

```
for pathname in $(find $search -type f -name "*" -print)
do
  basename=${pathname##*/}                    # replaces basename(1)
  dirname=${pathname%/*}                      # replaces dirname(1)
  ...
done
```

[1] `http://www.gnu.org/prep/standards.html`

Pattern matching in strings can be done with the comparison operator `=~` .

```
metacharacter='[~&|]'
if [[ "$pathname" =~ $metacharacter ]]
then
  # treat metacharacter
fi
```

Using POSIX regular expressions (`regex(7)`) is possible.[2]

9 SUID/SGID-Scripts

A shell script depends on user input, the process environment, the initialization files, the system utilities used etc. Shell languages are not well suited to write secure scripts because all of the above mentioned facilities (others as well) can be used to attack your system. Utilities may be vulnerable themselves.
There are a number of precautions which should be taken to run a SUID/SGID script [4, 6]. Here the most important without the claim for completeness:

- Execute the script from a directory where it can not be changed unauthorized.
- Check if the environment variable `BASH_ENV` is empty.
- Set **`umask`** to 077.
- Reset the environment variables `PATH` and `IFS` to secure values.
- Change to a safe working directory and validate this change.
- Use absolute path names for system utilities and data files.
- Check all return codes from system utilities.
- Signify the end of the option list with `--` .
- Quote all command line parameters (e.g. `"$1"`).
- Check the user input for shell metacharacters and other unwanted characters.
- Check user supplied pathnames (absolute/relative).
- Set the shell option `noclobber` to avoid overwriting existing files.
- Create temporary files in a save directory. Use `mktemp` (See section 6).

10 Testing

10.1 Syntax check

If a script with ***bash*** call option `-n` is executed, the script commands are read but not executed:

```
bash  -n  remove_ps.sh
```

Such call can be used for syntax check. However, only severe errors will be detected in this way. A mutilated key word (`cho` instead of **`echo`**) for example will not be detected, since it might also be the name of a program or a function.

[2] Most bash Shells are configured with the options `cond-command` and `cond-regexp`.

10.2 Test scope

In the development phase it is indispensable to organize a test environment with example files or example data of non-complex scope (e.g. in a directory tree organized for this purpose). This increases the process speed of the scripts and decreases the danger of making unintended changes to important data.

10.3 Use of echo

Commands causing a change, such as the deletion or renaming of files, should in test scope be first output as character strings by means of **echo** and checked. This is particularly advisable, when wildcards or recursive directory patterns are used. The instructions

```
for file in *.sh
do
  rm  "$file"
done
```

will immediately delete all files with the extension `.sh` in the overall directory tree. If the delete command is set initially into an **echo** instruction, the delete instructions are output in the same way as without **echo**.

```
  echo  "rm  \"$file\""
```

After verification **echo** can be removed.

10.4 Testing using *bash* options

Command line option	`set -o` Option	Meaning
`-n`	`noexec`	Commands are not executed, only syntax check (see 10.1)
`-v`	`verbose`	Outputs the lines of a script before execution.
`-x`	`xtrace`	Outputs the lines of a script after replacements.

Table 1: Options supporting the search for errors

If the lines

```
TMPFILE=$( mktemp /tmp/example.XXXXXXXXXX ) || exit 1
echo "program output" >> $TMPFILE
rm --force $TMPFILE
```

are executed with the options `-xv` by means of

```
bash -xv ./tempfile.sh
```

below output is generated:

```
TMPFILE=$( mktemp /tmp/example.XXXXXXXXXX ) || exit 1
 mktemp /tmp/example.XXXXXXXXXX
++ mktemp /tmp/example.XXXXXXXXXX
+ TMPFILE=/tmp/example.AVkuGd6796
echo "program output" >> $TMPFILE
+ echo 'program output'
rm --force $TMPFILE
+ rm --force /tmp/example.AVkuGd6796
```

The lines starting with + are generated by the -x option. The number of plus signs reflects the level of replacements. These options can be set in a script only for one section and then be cancelled again:

```
set -o xtrace          # --- xtrace on ---
for file in $list
do
  rm  "$file"
done
set +o xtrace          # --- xtrace off ---
```

10.5 Testing by means of trap

Pseudo signal	Trigger
DEBUG	The shell has executed a command.
EXIT	The shell terminates the script.

Table 2: Pseudo signals

The ***bash*** shell provides two pseudo signals which can be responded to by individual signal treatment. Figure 1 below shows the use of the two pseudo signals together with **trap** instructions. Figure 2 shows the output generated by this section.

```
1  #===  FUNCTION  ================================================================
2  #         NAME:  dbgtrap
3  #  DESCRIPTION:  monitor the variable 'act_dir'
4  #===============================================================================
5  function dbgtrap ()
6  {
7    echo "act_dir = \"$act_dir\""
8  }    # ----------  end of function dbgtrap  ----------
9
10 #-------------------------------------------------------------------------
11 #  traps
12 #-------------------------------------------------------------------------
13 trap 'echo "On exit : act_dir = \"$act_dir\""'  EXIT
14
15 trap dbgtrap DEBUG
16
17 #-------------------------------------------------------------------------
18 #  monitoring ...
19 #-------------------------------------------------------------------------
20 act_dir=$(pwd)
21 cd ..
22 act_dir=$(pwd)
23 cd $HOME
```

Figure 1: Example for the use of pseudo signals and **trap**

```
act_dir = ""
act_dir = "/home/mehner"
act_dir = "/home/mehner"
act_dir = "/home"
act_dir = "/home"
act_dir = "/home"
On exit : act_dir = "/home"
```

Figure 2: Output of script in figure 1

10.6 The debugger bashdb

The debugger `bashdb`[3] works with ***bash*** from version 3.0 and can be installed simply from the source package. It can interact also with the graphic debugger front end `ddd`[4]

11 Further sources of information

The most important source of information are the manuals for the actually installed version of the shell and the system utilities.
Occasionally technical journals publish articles on shell programming. Besides that there are a number of textbooks on shell programming. For questions concerning system programming and security a good point to start is [4, 6]. There are many platform oriented internet sites on security issues and new developments.

References

[1] Ken O. Burtch. *Linux Shell Scripting with Bash (Developer's Library)*. Sams, 2004.

[2] Mendel Cooper. Advanced Bash-Scripting Guide. http://www.tldp.org/LDP/abs/html/, 2010. Comprehensive tutorial with many examples, available in several formats. Well suited for additional online help and as reference work.

[3] FSF. Bash Reference Manual. http://www.gnu.org, Free Software Foundation, 12 2009. Bash shell, version 4.1. The official manual.

[4] Simson Garfinkel, Gene Spafford, and Alan Schwartz. *Practical Unix & Internet Security (3rd Edition)*. O'Reilly Media, 2003.

[5] Cameron Newham and Bill Rosenblatt. *Learning the bash Shell (3rd Edition)*. O'Reilly Media, 2005. Textbook; covers the features of Bash Version 3.0.

[6] David A. Wheeler. Secure Programming for Linux and Unix HOWTO, March 2003. Version v3.010.

[3] `http://bashdb.sourceforge.net`
[4] `http://www.gnu.org/software/ddd/`

12 Summary

No.	**Programming style**	Section
1	Line length max. 88 characters.	1
2	Indentation in accordance with nesting depth.	2
3	Each file has an introductory comment.	3.1
4	Consecutive line end comments start in the same column.	3.2
5	Interlinked instructions can be provided with a section comment.	3.3
6	A function always has a function comment.	3.4
7	Comments are short, concise and as precise as necessary.	3.5
8	A comment describes the purpose of an instruction.	3.5

No.	**Programming rules**	Section
1	Meaningful, self-documenting names shall be used for variables.	4.1
2	No constants must be included in the middle of program text.	4.2
3	The number and the validity of command line parameters must be verified.	5.1
4	The return values of programs and scripts have to be verified by means of the variable `$?`.	5.2
5	File extensions should indicate the contents.	6
6	For command line options the long form has to be used in scripts.	7

No.	**SUID/SGID-Scripts**	Section
1	Execute the script from a directory where it can not be changed unauthorized.	9
2	Check if the environment variable `BASH_ENV` is empty.	9
3	Set `umask` to 077.	9
4	Reset the environment variables `PATH` and `IFS` to secure values.	9
5	Change to a safe working directory and validate this change.	9
6	Use absolute path names for system utilities and data files.	9
7	Check all return codes from system utilities.	9
8	Signify the end of the option list with `--`.	9
9	Quote all command line parameters (e.g. `"$1"`).	9
10	Check the user input for shell metacharacters and other unwanted characters.	9
11	Check user supplied pathnames (absolute/relative).	9
12	Set the shell option `noclobber` to avoid overwriting existing files.	9
13	Create temporary files in a save directory. Use `mktemp` (See section 6).	9

No.	**Testing**	Section
1	Test using small test data amounts and *copies* of original data.	10.2
2	Critical commands for testing shall be first output and verified with `echo`.	10.3
3	Syntax check: ***bash***-command line switch `-n`.	10.4
4	Output script lines before execution: ***bash***-command line switch `-v`.	10.4
5	Output script lines after replacements: ***bash***-command line switch `-x`.	10.4
6	Tracing of single variables possible by using `trap`.	10.5