Bessere Shell-Scripte

Kieler Open Source und Linux Tage

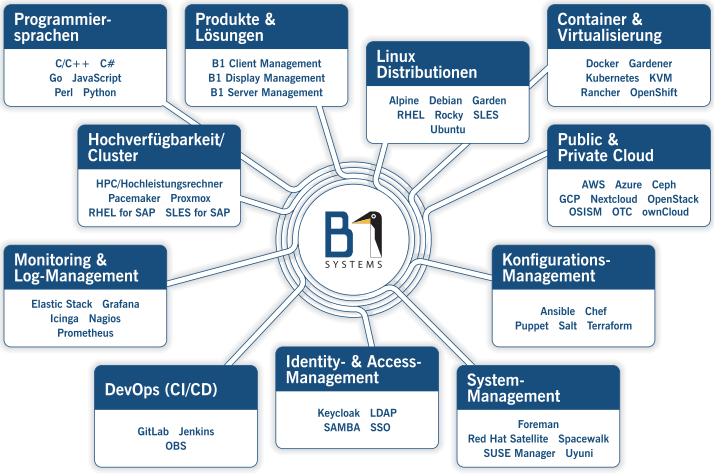
21. September 2024

Wolfgang Krüger
Linux Consultant & Trainer
B1 Systems GmbH
krueger@b1-systems.de

Vorstellung B1 Systems

- gegründet 2004
- spezialisiert auf Linux/Open Source-Themen
- national & international tätig
- ca. 150 Mitarbeiter:innen
- unabhängig von Soft- & Hardware-Herstellern
- Leistungsangebot:
  - Managed Service & Betrieb
  - Beratung & Consulting
  - Support
  - Training
  - Lösungen & Entwicklung
- Standorte in Rockolding, Köln, Berlin, Dresden & Jena

Schwerpunkte



Über mich

- seit 2000 freiberuflicher Trainer
- Programmierung PHP, Javascript, Java
- Objektorientierte Programmierung
- seit über 3 Jahren bei B1-Systems

Ziele

- bessere Fehlersuche
- bessere Struktur
- keine veraltete Syntax
- bessere Benutzbarkeit

Input – Positional Parameters

. . . wohl bekannt

```
#!/bin/bash

# Argumente ${1} - ${nn}
echo "Das 2. Argument ${2}"
echo "Alle Argumente: ${*}
echo "Die Anzahl der Argumente: ${#}"
echo "Der Name des Scriptes ${0}"
```

Tipp: Script mit unterschiedlichen Namen

Interessant ist vielleicht `${0}` – der Name des aufgerufenen Scriptes. Hiermit könnte das Script unter verschiedenen Namen aufgerufen werden.

Benutzereingaben lesen mit `read`

...wohl bekannt

```
read -p "Wie alt bist du?:"  INPUT
echo "Aha, du bist ${INPUT}"
```

Tipp: Prompt anzeigen

Die Option `-p` zeigt den angegebenen Text als Prompt an.

Input – Stream

...lesen des Datenstroms

```
read STREAM
UPPER=$( echo ${STREAM} |  tr 'a-z' 'A-Z' )
echo ${UPPER}
```

...und der Aufruf

```
echo "ich bin klein" | upper
ICH BIN KLEIN
```

Tipp: `read` im Datenstrom

`read` liest von der Standardeingabe. Damit kann das Script auch im Datenstrom verwendet werden.

Bedingungen – Alternativen zu test

Conditional Expressions

```
[[  anfang < ende  ]] ; echo ${?}
[[  45 -lt 5  ]] ; echo ${?}
```

Komplexe Syntax

```
[[ ( "${A}" -eq "0" || "${B}" -ne "0" ) && "${C}" -eq "0" ]] ; echo ${?}
```

Impliziter arithmetischer Kontext

```
[[ 'i=5, i+=2' -eq '3+4' ]] ; echo ${?}
```

Bedingungen – Numerischer Test

Arithmetic Evaluation

```
(( "${X}" < "10" )) ; echo ${?}
```

Rechnen auf der Shell

```
A=3
B=5
((C=A+B))

C=$(( 45*78 ))
```

Variablen

Variable als Integer deklarieren

```
declare -i A=23
declare -i B=5
declare -i C
C=${B}+4{A}
echo ${C}
28
```

Readonly Variable = Konstante

```
declare -r fix=42
readonly fix_too=23
fix=55
bash: fix: Schreibgeschützte Variable.
```

Funktionen

Funktionen mit Rückgabewerten

```
codetest() {
  echo "Hallo";
  return 1;
}
```

Funktionen – lokale Variablen

```
function xy {
  X=23
  local Y=42
}
```

Funktionen – Hilfe

Hilfe anzeigen

```
usage() {
  echo "Aufruf: myscript file"
  echo "Bitte geben Sie eine Datei an"
  exit 0
}
```

Funktionen – Fehlerbehandlung

Fehlerbehandlung

```
error_handle() {
    if [[ -n "${2}" ]] ; then
        echo ${2} 1>&2
    fi
    exit ${1}
}
```

Strukturieren – aber wie?

- EVA-Prinzip (banal):
  - **E**ingabe (Quelle, Tests, Meldungen)
  - **V**erarbeitung
  - **A**usgabe (STDOUT, STDERR, Exit-Code)
- MVC Entwurfsmuster (professionell):
  - **M**odel (Verarbeitung)
  - **V**iew (Ausgabe)
  - **C**ontroller (Eingabe, Ablauf)

Grundlegende Struktur

- #!SHEBANG
- Dokumentations-Kommentar
- Konfigurations-Variablen, Konstanten
- Variablen-Initialisierung
- Funktions-Definitionen
- Code (Aufruf der Funktion `main`)

Tipps zur Fehlersuche 1/4

Befehl zum Testen ausgeben

```
echo touch ${file_list}
```

Variable zum Test ausgeben – mit `read` das Script pausieren

```
for NUMBER in $(seq 10) ; do
    echo $NUMBER
    read
done
```

Script an beliebiger Stelle beenden

```
exit
```

Tipps zur Fehlersuche 2/4

Shell in den Debug-Modus setzen – global

```
#!/bin/bash -x

pwd
```

Shell in den Debug-Modus setzen – lokal

```
set -x
date
set +x
```

Tipps zur Fehlersuche 3/4

Shell in den nounset-Modus setzen

```
set -u
echo $XIZ
```

Shell in den errexit-Modus setzen

```
set -e
ls notexists
```

Shell in den pipefail-Modus setzen

```
set -o pipefail
 echo hallo | cat | grep --nix  h | tr h H
echo $?
```

Tipps zur Fehlersuche 4/4

Einen Debugmodus definieren

```
declare -r DEBUG="0"

if "${DEBUG}" ; then
  debug_print_vars
fi
```

Explizite Fehlerbehandlung

Explizites Testen von Variablen

```
if [[ ${TEST+defined} ]] ; then echo "is defined" ; fi
```

Explizite Fehlerbehandlung

```
grep -q 'notexists' filename || error_handler
grep -q 'notexists' filename ; exit_code=$?
grep -q 'notexists' filename || true
```

Obsolete and deprecated syntax

Command Substitution

verwende `$(command)` anstatt Backticks `` `...` ``

`test`, [...], and [[...]]

verwende [[...]] anstatt von [...], `test` und `/usr/bin/[`

Arithmetic

- verwende immer ((...)) oder $((...))
- verwende nie die $[...] Syntax, den `expr`-Befehl oder das `let`-Built-in
- verwende niemals `eval`

Code Style Guide

- Einrückung mit zwei Leerzeichen
- Umbrechen von langen Zeilen
  - maximale Zeilenlänge 80 Zeichen
  - Einrückung der weiteren Zeilen
- Kryptische Konstrukte
  - wenn sie nicht zu 100 % benötigt werden – vermeiden
  - wenn Sie verwendet werden müssen, kommentieren was das „Monster“ tut
- sinnvoll verlassen
  - Exitcode 0 (Null) wenn alles in Ordnung ist
  - Exitcode 1 – generell „ungleich Null“ – wenn es einen Fehler gab
  - normale Ausgaben nach STDOUT
  - Fehler-, Warn- und Diagnosemeldungen nach STDERR

Code Style Guide

- im Zweifelsfall: Konsequent sein
- VIM Mode Lines
  - `# vim: ts=2 sts=2 sw=2 expandtab ai`
  - `set modeline`
  - `:verbose set modeline? modelines?`

Parameter Expansion

```
${parameter:offset:length}  # Substring Expansion
${#parameter}               # Parameter length
${parameter#word}           # Remove matching prefix pattern
${parameter%word}           # Remove matching suffix pattern
${parameter/pattern/string} # Pattern substitution
${parameter^pattern}        # Case modification uppercase
${parameter,pattern}        # Case modification lowercase
${parameter:+word}          # Alternate Value
```

Parameter Expansion

Beispiele

```
STRING="Ein Test"
echo ${#STRING}          # 8
echo ${STRING:1:2}       # in
echo ${STRING%Test}      # Ein
echo ${STRING#Ein}       # Test
echo ${STRING^^[aeiou]} # EIn TEst
```

Parameter $0 – basename

```
FULLNAME=/usr/bin/ls
basename=${FULLNAME##*/}
pathname=${FULLNAME%${basename}}
```

Optionen – kurz

Beispiel

```
while getopts "abc:d:" opt 2>/dev/null ; do
  case ${opt} in
     a) echo "argument -a" ;;
     b) echo "argument -b" ;;
     c) echo "argument -c ${OPTARG}" ;;
     d) echo "argument -d ${OPTARG}" ;;
     ?) echo "Fehler in den Optionen" >&2 ;;
  esac
done
```

Optionen – lang

Die langen Optionen könnten in einer eigenen Schleife abgefragt werden.

Beispiel

```
while [[ $# -gt 0 ]] ; do
  key="$1"
  case "$key" in
  --value) value="${2}"; shift 2;;
  --value=*) value=${key#*=} ; shift;;
  --test) TEST=true; shift;;
done
```

Optionen – lang und kurz Lösung 1

- kurze und lange Optionen werden mit ODER verknüpft abgefragt
- Nachteil: kurze Optionen können nicht wie gewohnt zusammengefasst werden

Beispiel

```
while [[ $# -gt 0 ]] ; do
  key="${1}"
  case "${key}" in
  -v|--value) value="${2}"; shift 2;;
  -v=*--value=*) value=${key#*=} ; shift;;
  t|--test) TEST=true; shift;;
done
```

Optionen – lang und kurz Lösung 2

Bash Builtin `getopts` kann verwendet werden, indem ein Bindestrich gefolgt von einem Doppelpunkt in die Optspec eingefügt wird.

Beispiel

```
while getopts hv-: optchar; do
  case "${optchar}" in
    -)
      case "${OPTARG}" in
        loglevel) val="${!OPTIND}"; OPTIND=$(( $OPTIND + 1 )) ;;
        loglevel=*) val=${OPTARG#*=} ; opt=${OPTARG%=$val} ;;
      esac
  esac
done
```

Optionen – lang und kurz Lösung 3

GNU `getopt` bietet Unterstützung für lang benannte Kommandozeilenoptionen.

Beispiel

```
ARGS=$(getopt -o vtm: --long verbose,test,memory: -- "$@")
set -- "${ARGS}"
```

dialog

Einfache Ausgaben

```
dialog --msgbox "Das ist Dialog" 5 50
```

Einfache Ausgaben

Abbildung: Dialog Message Box

dialog – einfache Eingaben

Bei einfachen Ja/Nein Entscheidungen verwendet `dialog` den Rückgabewert.

```
dialog --yesno "Magst du Linux?" 5 50 ; linux=${?}
clear
echo ${linux}
```

dialog – Eingaben

- `dialog` gibt die Benutzereingaben auf STDERR aus, da STDOUT bereits von `ncurses` zur Aktualisierung des Bildschirms verwendet wird
- `STDERR` und `STDOUT` können durch Rerouting getauscht werden

```
day=$(date '+%d'); month=$(date '+%m'); year=$( date '+%Y')
birthday=$(dialog  --date-format '%Y-%m-%d' \
                   --calendar 'Wann ist dein Geburtstag?' 5 50 \
                   ${day} ${month} ${year} \
                   3>&1 1>&2 2>&3 3>&- )

clear
echo $birthday
```

Quellen & Links

- https://mywiki.wooledge.org/BashFAQ/105
- https://web.archive.org/web/20161218202153/http://wiki.bash-hackers.org/start
- https://google.github.io/styleguide/shellguide.html
- https://github.com/rawiriblundell/wiki.bash-hackers.org/blob/main/scripting/style.md
- https://github.com/rawiriblundell/wiki.bash-hackers.org/blob/main/scripting/obsolete.md
- https://web.archive.org/web/20161218202153/http://wiki.bash-hackers.org/scripting/obsolete
- https://stackoverflow.com/questions/402377/using-getopts-to-process-long-and-short-command-line-options
- https://stackoverflow.com/questions/29222633/bash-dialog-input-in-a-variable
- https://stackoverflow.com/questions/8515411/what-is-indirect-expansion-what-does-var-mean
- https://vim.fandom.com/wiki/Modeline_magic

Vielen Dank für Ihre Aufmerksamkeit!

Bei weiteren Fragen wenden Sie sich bitte an info@b1-systems.de oder +49 (0)8457 - 931096