FREE eBook

LEARNING sh

Free unaffiliated eBook created from
Stack Overflow contributors.

#sh

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: sh

It is an unofficial and free sh ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official sh.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with sh

Remarks

`sh` is not a single shell. Rather, it is a specification with the POSIX operating system standard for how a shell should work. A script that targets this specification can be executed by any POSIX-compliant shell, such as

- `bash`
- `ksh`
- `ash` and its derivatives, such as `dash`
- `zsh`

In a POSIX-compliant operating system, the path `/bin/sh` refers to a POSIX-compliant shell. This is usually a shell that has features not found in the POSIX standard, but when run as `sh`, will restrict itself to the POSIX-compliant subset of its features.

References

- Standard `sh`
- The FreeBSD `sh(1)` man-page
- The NetBSD `sh(1)` man-page
- The OpenBSD `sh(1)` man-page
- The Illumos `sh(1)` man-page (`ksh93(1)`)

Examples

Hello, world!

With `echo`:

```
$ echo Hello, world!
Hello, world!
```

With `printf`:

```
$ printf 'Hello, world!\n'
Hello, world!
```

As a file:

```
#!/bin/sh
printf '%s\n' 'Hello, world!'
```

Echo Portability

```
$ for shell in ash bash dash ksh ksh93 zsh; do
>     $shell -c "echo '\\\\'$shell'\\\\'"
> done
\\ash\\
\\bash\\
\dash\
\pdksh\
\\ksh93\\
\zsh\
```

'echo' can only be used consistently, across implementations, if its arguments do not contain any backslashes (reverse-solidi), and if the first argument does not start with a dash (hyphen-minus). Many implementations allow additional options, such as `-e`, even though the only option allowed is `-n` (see below).

From POSIX:

> If the first operand is -n, or if any of the operands contain a character, the results are implementation-defined.

Read Getting started with sh online: https://riptutorial.com/sh/topic/3300/getting-started-with-sh

Chapter 2: Arithmetic Expansion

Remarks

Numbers in arithmetic expansions must match the following ERE:

```
[-+]?(0[0-7]+|[1-9][0-9]*|0[Xx][0-9A-Fa-f]+)
```

Arithmetic expressions support signed integer operators, comparisons, Boolean expressions, assignments, and ternary expressions from C.

Resources

- Arithmetic expansion in POSIX
- Operator precedence

Examples

Line Count

```
i=0
while read -r line; do
        i=$((i+1))
done < file
echo $i
```

With a file containing:

```
Alpha
Beta
Gamma
Delta
Epsilon
```

The above script prints: `5`

Parameter Expansion

Loop `n` times:

```
while [ $((i=${i:=0}+1)) -le "$n" ]; do
    echo line $i
done
```

Output for `n=5`:

```
line 1
line 2
line 3
line 4
line 5
```

Manipulating decimals:

```
$ i=3.14159; echo $((${i%.*}*2))
6
$ i=3.14159; echo $((${i#*.}*2))
28318
```

Ternery Expressions

Absolute value:

```
$ for n in -8 -2 0 3 4; do
>     echo $((n<0?-n:n))
> done
8
2
0
3
4
```

Fix variable range:

```
$ min=2
$ max=4
$ for n in 1 2 3 4 5; do
>     echo $((n<min?min:n>max?max:n))
> done
2
2
3
4
4
```

Is a Power of 2

```
$ ispow2() { return $((!($1!=0&&($1&$1-1)==0))); }
$ i=0
$ while [ $i -lt 100 ]; do
>     if ispow2 $((i=i+1)); then
>         echo $i
>     fi
> done
1
2
4
8
16
32
```

```
64
```

`$1!=0` 0 is not a power of 2.

`($1&$1-1)==0` Unset the lowest bit. If it was the only bit then the number was a power of 2.

The additional `!` was for correcting the value to what the shell expects, which is the opposite of the conventional true/false values (zero for true and non-zero for false, vs zero for false and non-zero for true).

Read Arithmetic Expansion online: https://riptutorial.com/sh/topic/6223/arithmetic-expansion

Chapter 3: IO Redirection

Introduction

Generally a command takes inputs from terminal and outputs back to terminal. Normally a command reads input from keyboard and outputs result to the screen. Here is the importance of Input/Output Redirection

Syntax

- [fd]<file
- [fd]<&fd
- [fd]<&-
- [fd]>file
- [fd]>&fd
- [fd]>&-
- [fd]>|file
- [fd]>>file
- [fd]<>file
- [fd]<<[-] word
  ...
  word

Remarks

Resources

- The POSIX 'Shell Command Language' section on 'Redirection'

Examples

Output Redirection

Usually output of a command goes to the terminal. Using the concept of Output redirection, the output of a command can be redirected to a file. So insted of displaying the output to the terminal it can be send to a file. '>' character is used for output redirection.

```
$ pwd > file1
$ cat file1
/home/cg/root
```

In the above example, the command the output 'pwd' of the command is redirected to a file called 'file1'.

Input Redirection

The commands normally take their input from the standard input device keyboard. Using Input redirection concept, we can have their input redirected from a file. To redirect standard input from a file instead of the keyboard, the '<' character is used.

```
$ cat file1
monday
tuesday
wednsday
thursday
friday
saturday
sunday
```

The above is the content of file1

```
$ sort < file1
friday
monday
saturday
sunday
thursday
tuesday
wednsday
```

here insted of taking input from keyboard, we redirected it from the file1 and sort it in ascending order.

Read IO Redirection online: https://riptutorial.com/sh/topic/9345/io-redirection

Chapter 4: Job Control

Examples

Pause, run in background, run in foreground

Let's create a process which is rather long to complete :

```
$ sleep 1000
```

To pause the process, type Ctrl + Z :

```
^Z
[1]+  Stopped                 sleep 1000
```

You can use `jobs` to see the list of processes running or stopped in the current terminal :

```
$ jobs
[1]+  Stopped                 sleep 1000
```

To bring back a job on the foreground, use `fg` with the id written between brackets in the list provided by `jobs` :

```
$ fg 1
sleep 1000
```

When a job is stopped, you can run it in background with the command `bg` with the same id :

```
$ bg 1
[1]+ sleep 1000 &
```

And then see it in the list of jobs in the current terminal :

```
$ jobs
[1]+  Running                 sleep 1000 &
```

To directly run a job in background, finish the command with `&` :

```
$ jobs
[1]+  Running                 sleep 1000 &
$ sleep 5000 &
[2] 6743
$ jobs
[1]-  Running                 sleep 1000 &
[2]+  Running                 sleep 5000 &
```

List, wait and stop processes

To get a list of the processes running in the current terminal, you can use `ps` :

```
$ sleep 1000 &
$ ps -opid,comm
  PID COMMAND
 1000 sh
 1001 sleep
 1002 ps
```

To kill a running process, use `kill` with the process ID (PID) indicated by `ps`:

```
$ kill 1001
$ ps -opid,comm
 PID COMMAND
1000 sh
1004 ps
```

To wait for a process to terminate, use the `wait` command :

```
$ sleep 10 && echo End &
$ ps -opid,comm
 PID COMMAND
1000 sh
1005 sh
1006 sleep
1007 ps
$ wait 1005 && echo Stop waiting
End
Stop waiting
```

First, we run a process with PID 1005 in background which will print "End" before ending. Then, we wait for this process to finish, and print "Stop waiting". The output shows "End", meaning the process with PID 1005 is complete, then "Stop waiting", showing the wait command is complete.

Read Job Control online: https://riptutorial.com/sh/topic/6932/job-control

Chapter 5: Quoting

Remarks

References

- The POSIX 'Shell Command Language' section on 'Quoting'

Examples

Single-Quotes

Single-quotes are literal strings, and the lack of escape characters means that the only character that can not occur inside of a single-quoted string is a single-quote.

```
$ echo '$var \$var \\$var \\\$var'
$var \$var \\$var \\\$var
$ echo '"quoted string"'
"quoted string"
$ echo 'var=$(echo $var)'
var=$(echo $var)
```

Double-Quotes

Double-quotes preserve all characters other than `"` terminator, `$` expansions, `` ` `` command substitutions, and `\` escapes of any of these characters (and newline removal). Note that the literal `\` is preserved unless followed by a special character.

General escapes:

```
$ printf "\"quoted string\"\\n"
"quoted string"
$ printf "\`\`quoted string''\n"
``quoted string''
$ printf "four\\\\nthree\\\ntwo\\none\n"
four\nthree\ntwo
one
$ echo "var=\`echo \$var\`"
var=`echo $var`
$ echo "var=\$(echo \$var)"
var=$(echo $var)
```

Variable expansion:

```
$ var=variable echo "$var \$var \\$var \\\$var"
variable $var \variable \$var
```

Command substitution:

```
$ var=variable echo "var=`echo $var`"
var=variable
$ var=variable echo "var=$(echo $var)"
var=variable
```

Removing newlines:

```
$ echo "multi\
> -line"
multi-line
```

Escaping

`\` escapes preserve the following character value, unless the following character is a newline in which case both the `\` and the newline are removed.

Escaping special characters:

```
$ echo \"quoted text\"
"quoted text"
$ echo \`\`quoted text\'\'
``quoted text''
$ echo 'single-quotes inside of a '\''single-quoted'\'' string'
single-quotes inside of a 'single-quoted' string
$ printf format\ with\ %s spaces
format with spaces
$ printf %s\\n \$var
$var
```

Removing newlines:

```
$ echo multi\
> -line
multi-line
```

Read Quoting online: https://riptutorial.com/sh/topic/5947/quoting

Chapter 6: Test

Syntax

- **test**
- **test** [!] [-n | -z] *string*
- **test** [!] { -b | -c | -d | -e | -f | -g | -h | -L | -p | -r | -S | -s | -u | -w | -x } *file*
- **test** [!] -t *fd*
- **test** [!] *string* { = | != } *string*
- **test** [!] *integer* { -eq | -ne | -gt | -ge | -lt | -le } *integer*
- **[ ]**
- **[** [!] [-n | -z] *string* **]**
- **[** [!] { -b | -c | -d | -e | -f | -g | -h | -L | -p | -r | -S | -s | -u | -w | -x } *file* **]**
- **[** [!] -t *fd* **]**
- **[** [!] *string* { = | != } *string* **]**
- **[** [!] *integer* { -eq | -ne | -gt | -ge | -lt | -le } *integer* **]**

Remarks

If `test(1)` is run without any arguments it returns false.

Reference

- Standard `test(1)`
- The FreeBSD `test(1)` man-page
- The NetBSD `test(1)` man-page
- The OpenBSD `test(1)` man-page
- The Illumos `test(1)` man-page
- The GNU Coreutils online manual section on `test(1)`

Examples

Multiple Expressions

Though it is an obsoleted part of the XSI standard, many implementations still support multiple expressions with Boolean operators and parenthesis.

The (obsolete) operators are listed below with decreasing precedence.

```
( expression )
expression -a expression
expression -o expression
```

Using these (obsolete) operators, a complex shell expression:

```
if [ "$a" -gt 0 ] && { [ "$b" -ne 2 ] || [ "$b" -e 0 ]; }
then ...
fi
```

Could be written with one invocation of `test(1)`:

```
if [ "$a" -gt 0 -a '(' "$b" -ne 2 -o "$c" -ne 0 ')' ]
then ...
fi
```

Read Test online: https://riptutorial.com/sh/topic/7683/test

Chapter 7: The `read` command

Examples

Read a line verbatim

```
$ IFS= read -r foo <<EOF
>     this is a \n line
>EOF
$ printf '%s\n' "$foo"
    this is a \n line
```

Read a line, stripping leading and trailing whitespace

```
$ read -r foo <<EOF
>    this is a line
>EOF
$ printf '%s\n' "$foo"
this is a line
```

Read The `read` command online: https://riptutorial.com/sh/topic/3954/the--read--command

Credits

S. No	Chapters	Contributors
1	Getting started with sh	chepner, Community, Dunatotatos, kdhp
2	Arithmetic Expansion	kdhp
3	IO Redirection	Anand C, kdhp
4	Job Control	Dunatotatos, kdhp
5	Quoting	kdhp
6	Test	kdhp
7	The `read` command	chepner