

# Unix Korn Shell Scripting Reference Sheet

**Remember:** Unix is case sensitive.

## Initialization files

<code>/etc/profile</code>	Contains the system default shell start-up commands.
<code>\${Home}/.profile</code>	Contains any options and shell functions you want available to your personal operating environment.
<code>\${ENV}</code>	Contains any options and shell functions you want available everywhere. Common names are <code>\$HOME/.kshrc</code> and <code>\$HOME/.env</code> . Executed each time a child Korn shell is created by a <code>fork(2)</code> system call.

## Special Characters

<code>\$</code>	Indicates the beginning of a shell variable name
<code> </code>	Pipe standard output to the next command
<code>#</code>	1) Comment character. 2) Also used as <code>#!</code>
<code>&amp;</code>	Executes a process in the background
<code>?</code>	Matches one character
<code>*</code>	Matches one or more characters
<code>&gt;</code>	Output redirection operator
<code>&lt;</code>	Input redirection operator
<code>&gt;&gt;</code>	Output redirection operator (to append to a file)
<code>&lt;&lt;</code>	Wait until following end-of-input string (HERE operator)
<code>space</code>	Delimiter between two words
<code>\</code>	1) Escape character (stops the shell from interpreting the succeeding character as a special character. 2) Line continuation character.
<code>+</code>	1) Pattern matching operator 2) Arithmetic operator inside a let command.
<code>;</code>	Separates commands if used on the same line.
<code>.script.sh</code>	Runs the script in the current shell. Equivalent to typing all the lines at the command line.

## Double Quotes, Forward Tics and Back tics

<code>"..."</code>	1) Allows variable and command substitution in strings with embedded spaces (double quote). 2) Protects all special characters except <code>;</code> , <code>`</code> , <code>\</code>
<code>'...'</code>	1) Does not allow variable and command substitution in strings (single quote). 2) Protects all special characters except itself.
<code>`...`</code>	Command substitution (The backquote or backtick) (replaces a string with the output of the executed command) also achieved using <code>\$(command)</code>

## Brackets

<code>()</code>	1) Groups commands together (See command control). 2) Used in pattern lists (See section on pattern lists) 3) Runs the enclosed command in a subshell.
<code>(( ))</code>	Evaluate and assign value to variable and do math in a shell
<code>[ ]</code>	1) A range of characters (See section on wildcard expansion). 2) Array subscript. 3) Same as test command. There must be spaces between the brackets and the condition.
<code>[[ ]]</code>	Conditional expression.
<code>\$()</code>	Command substitution
<code>\$( )</code>	Evaluate the enclosed expression
<code>{ }</code>	1) Parameter substitution. 2) Runs the enclosed command in a subshell.

## Command controls

<code>#!cmd</code>	If this the first line of the script, the script is run under <code>cmd</code> . Usually, <code>cmd</code> is <code>/bin/ksh</code> etc. Otherwise it is a comment.
<code>cmd &amp;</code>	Runs <code>cmd</code> in the background.
<code>cmd   &amp;</code>	Runs <code>cmd</code> in the background asynchronously with the parent shell. The parent shell then uses <code>read -p</code> and <code>print -p</code> to communicate with <code>cmd</code> .
<code>cmd1 ; cmd2</code>	Command separator. Runs <code>cmd1</code> then runs <code>cmd2</code> .
<code>(cmds)</code>	Command grouping. Runs <code>cmds</code> in a child shell.
<code>{cmds ; }</code>	Command grouping. Runs <code>cmds</code> in a child shell.

## File Name Generation

### Wildcard Expansion

<code>*</code>	Matches any string including null string.
<code>?</code>	Matches any single character.
<code>[ch1]</code>	Matches any of the enclosed characters <code>chl</code> .
<code>[n-z]</code>	Matches characters n through to z.
<code>[a,z]</code>	Matches characters a or z.
<code>[az]</code>	Matches characters a or z.

## Variables

### Environment Variables set by the Korn shell

<code>\$ERRNO</code>	Return code from the most recently set system call.
<code>\$LINENO</code>	Line number of the current script (or shell function).
<code>\$OPTARG</code>	Value of the last option argument processed by the <code>getopts</code> command.
<code>\$OPTIND</code>	Index position of the last option argument processed by the <code>getopts</code> command.
<code>\$PWD</code>	Working directory.

### Environment Variables used by the Korn shell

<code>\$CDPATH</code>	Search path for the <code>cd</code> built-in command.
<code>\$COLUMNS</code>	Number of columns in the user's terminal display.
<code>\$ENV</code>	Path name to the script executed when the shell is invoked.
<code>\$FPATH</code>	Search path for function definitions.
<code>\$HOME</code>	Set to the name of your home directory.
<code>\$IFS</code>	Internal field separator. The default IFS is set to a space, tab stop and carriage return.
<code>\$LINES</code>	Number of lines in the user's terminal display.
<code>\$MAIL</code>	Name of the mail file to check for incoming electronic mail.
<code>\$MAILPATH</code>	List of the mail files to check for incoming electronic mail. This variable overrides the <code>\$MAIL</code> variable
<code>\$MAILCHECK</code>	Specifies how often your mailbox is checked for new mail. The default value is 600 (seconds).
<code>\$PS1</code>	Primary prompt string, <code>\$</code> by default. You can change this to whatever you want.
<code>\$PS2</code>	Secondary prompt, <code>&gt;</code> by default. This is the prompt you will see if you enter a command that requires more input than you have provided. This could happen if, for example, you continue a line or forget to close quotes.
<code>\$PS3</code>	Primary prompt string for the <code>select</code> loop. Initially set to <code># ?</code> .
<code>\$PS4</code>	Trace mode line identifier. Initially set to <code>+</code> .
<code>\$PATH</code>	List of directories to search for an executable command.
<code>\$SHELL</code>	The current unix shell in session.
<code>\$TERM</code>	The type of terminal you are using.
<code>\$USER</code>	Username of user.

### Built-in variables or automatic parameters

<code>\$0</code>	Set to the name of the currently executing script. Often used in error messages to identify the script in which the error occurred.
<code>\$#</code>	Set to the number of arguments passed to the script. Usually tested at the beginning of a script to check that the user has passed the correct number of arguments.
<code>\$*</code>	Contains a list of all the arguments passed to the script in the form <code>"\$1 \$2 ... "</code> .
<code>\$@</code>	Contains a list of all the arguments passed to the script in the form <code>"\$1" "\$2" ...</code>
<code> \$?</code>	The return code of the last executed command. If the command was successful, <code> \$?</code> is 0.
<code> \$\$</code>	The process number (PID) of the current shell. Since no two processes can run simultaneously with the same PID it is a unique number in the short term. Used to generate unique filenames within a script.
<code> \$!</code>	PID of the last process placed in the background. This number is needed to kill the process.
<code> \$-</code>	Contains the shell options that are currently set.
<code> \$_</code>	Absolute path name of the shell or script being executed. After execution, the shell or script path name is set to the last argument of the previous command.

## Positional Parameters

\$1, \$2 ... First, second *etc* taken from the position of the output.

## Parameter Substitution

Parameter substitution is signified by { }.

The General form is: \${<parameter\_name>[:<arguments>]} where <> is a substitution and [ ] is optional.

`\${pattern}`	Executes <i>pattern</i> in a child shell.
`\${param}`	Braces are used when a parameter substitution conflict may arise i.e the braces are text separators
`\${#param}`	If <i>param</i> is *or @, substitutes the number of positional parameters; otherwise, substitutes the length of the value of <i>param</i> .
`\${#identifier[*]}`	Substitutes the number of elements in the array <i>identifier</i> .
`\${param:-word}`	If <i>param</i> has a value that value is substituted. If not, word is substituted.
`\${param:=word}`	If <i>param</i> has a value that value is substituted. If not, word is assigned to <i>param</i> and substituted.
`\${param:+word}`	The exact opposite of :- If <i>param</i> has a value word is substituted. If not, a null is substituted. This can be used to temporarily override a variable.
`\${param:?word}`	If <i>param</i> has a value that value is substituted. If not, word is printed to <b>stdout</b> and the shell exits.
`\${param#pattern}` `\${param##pattern}`	If the shell pattern matches the beginning value of <i>param</i> , the value of this substitution is the value of <i>pattern</i> , with the matched portion deleted. In the first form, the smallest matching pattern is deleted; in the second, the largest matching pattern is deleted.
`\${param%pattern}` `\${param%%pattern}`	If the shell pattern matches the ending value of <i>param</i> , the value of this substitution is the value of <i>pattern</i> , with the matched portion deleted. In the first form, the smallest matching pattern is deleted; in the second, the largest matching pattern is deleted.

## Pattern Lists

?(pattern-list)	Optionally matches anyone of the given patterns.
*(pattern-list)	Matches zero or more occurrences of the given patterns.
+(pattern-list)	Matches one or more occurrences of the given patterns.
@(pattern-list)	Matches exactly one of the given patterns.
!(pattern-list)	Matches anything except the given patterns.

## Flow Control

### If

```
if [ $? -eq 0 ]
  then grep blob
elif [ $? -eq 1 ]
  then grep blob
else
  grep blob
fi
```

### For

```
for loop_variable in argument_list
do grep loop_variable
done
```

### While

```
while test_true
do grep blob
done
```

### Until

```
until test_true
do grep blob
done
```

## Case Statement

```
case $VARIABLE in
match1) commands to execute for 1;;
match2) commands to execute for 2;;
*) (Optional any other value)
  commands to execute for no match;;
```

### esac

<b>break</b>	Used to terminate the execution of the entire loop, after completing the execution of all the lines of code up to the break statement.
<b>continue</b>	Used to transfer control to the next set of code, but it continues execution of the loop.
<b>exit</b>	Exits the script . An integer may be added to the command, which will be sent as the return code.
<b>return</b>	Used in a function to send data back, or return a result to the calling script

## Comparisons

### String Comparison

```
string=pattern      string matches pattern.
string!=pattern     string does not match pattern.
string1<string2     string1 comes before string2 based on the
                    ASCII values for the characters.
string1>string2     string1 comes before string2 based on the
                    ASCII values for the characters.
```

## Number Comparison

-eq Equal  
-ne Not equal  
-ge Greater than or equal too  
-le Less than or equal too  
-gt Greater than  
-lt Less than

## File Comparison

*file1* -ef *file2* Both *file1* and *file2* exist and refer to the same file  
*file1* -nt *file2* *file1* is newer than *file2*.  
*file1* -ot *file2* *file1* is older than *file2*.

## Arrays

Indexing starts at zero.

## Conditional Logic on Files

-a file exists.  
-b file exists and is a block special file.  
-c file exists and is a character special file.  
-d file exists and is a directory.  
-e file exists (just the same as -a).  
-f file exists and is a regular file.  
-g file exists and has its *setgid*(2) bit set.  
-G file exists and has the same group ID as this process.  
-k file exists and has its *sticky* bit set.  
-L file exists and is a symbolic link.  
-n *string* length is not zero.  
-o Named *option* is set on.  
-O file exists and is owned by the user ID of this process.  
-p file exists and is a first in, first out (FIFO) special file or named pipe.  
-r file exists and is readable by the current process.  
-s file exists and has a size greater than zero.  
-S file exists and is a socket.  
-t file descriptor number *fdes* is open and associated with a terminal device.  
-u file exists and has its *setuid*(2) bit set.  
-w file exists and is writable by the current process.  
-x file exists and is executable by the current process.  
-z *string* length is zero.

## Functions

```
function function_name
{commands to execute}
```

or

```
function_name ()
{commands to execute}
```

## Declaring a shell

#!/usr/bin/sh	#!/bin/sh	Bourne
#!/usr/bin/ksh	#!/bin/ksh	Korn
#!/usr/bin/csh	#!/bin/csh	C
#!/usr/bin/bash	#!/bin/bash	Bourne-Again

## Here Document

```
program_name <<LABEL Program_Input_1 Program_Input_2
LABEL
```

## Mathematical Operators

++ Auto-increment, both prefix and postfix.  
-- Auto-decrement, both prefix and postfix.  
+ Unary plus.  
- Unary minus.  
! Logical negation;binary inversion (one's complement).  
\* Multiplication  
/ Division.  
% Modulus (remainder)  
== Equality.  
!= Inequality.  
<= Less than or equal too.  
>= Greater than or equal too.  
< Less than.  
> Greater than.  
<< Bitwise left shift.  
>> Bitwise right shift  
& Bitwise AND.  
^ Bitwise exclusive OR.  
| Bitwise OR

## Compound Logical expressions

(*expression*) True if *expression* is true.  
!*expression* NOT *expression*. True if expression is false.  
*exp1 -a exp2* Logical AND  
*exp1&&exp2* Logical AND  
*exp1-oexp2* Logical OR  
*exp1||exp2* Logical OR

## Intrinsic Functions

abs Absolute value  
sqrt Square root  
exp Exponential function  
int Integer part of floating point number  
cos, sin cosine, sine *etc*

## Interrupt Handling

When a program is terminated prematurely, we can catch an exit signal. This signal is called a *trap*. To see the entire list of supported signals for your operating system type: `kill -l`  
That is: `kill (-e11)` The following command will print a message to the screen and make a clean exit on signals 1,2,3 and 15:  
`trap 'echo "\nExiting on a trapped signal" exit;' 1 2 3 15`

## Exit Signals

0 - Normal termination, end of script.  
1 SIGHUP Exit Hangup, line disconnected. If `nohup` was used to execute the script, it would be prevented from hangup.  
2 SIGINT Exit Terminal interrupt, usually CONTROL-C.  
3 SIGQUIT Core Quit key, child processes to die before terminating  
9 SIGKILL Exit Kill -9 command, **cannot** trap this type of exit status.  
11 SIGSEGV Core Segmentation Fault  
15 SIGTERM Exit Kill command's default action  
24 SIGTSTP Stop, usually CONTROL-Z.

## Command: set -o options

Typing the command `set -o` will tell you the options which can be set in the shell. By inputting `set -option` you are turning a setting on and by using `set +option` you are turning a setting off. A summary of the most useful options are given.

**all export** Automatically exports all subsequently defined parameters. The same as `set -a`  
**bgnice** Runs all background jobs in nice mode.  
**emacs** sets in-line editor to emacs.  
**errexit** If a command has a non-zero exit status, then executes the ERR trap (signal 7). The same as `set -e`.  
**gmacs** sets in-line editor to gmacs.  
**ignoreeof** Does not exit shell on Ctrl D.  
**keyword** Places all parameter assignments in the current environment. The same as `set -k`.  
**markdirs** Marks all directories with a trailing `/`.  
**monitor** Runs background jobs in a separate process group. The same as `set -m`.  
**noclobber** Does not allow destruction of existing file contents through shell redirection of output Use `>|` to override noclobber.  
**noexec** Reads commands and checks for syntax errors, but does not execute them. Ignored in interactive shells. The same as `set -m`.

**noglob** Inhibits file name expansion. Treats all characters as literal text. The same as `set -n`. This is useful for debugging  
**nolog** Does not save function definitions in the history file.  
**nounset** Treats unset parameters as an error, if they are unset during substitution. The same as `set -u`.  
**privileged** Disabled processing of `${Home}/.profile`. Uses `/etc/suid_profile` in place of `${ENV}`. The same as `set -p`.  
**trackall** Each command becomes a tracked alias when first encountered. The same as `set -h`.  
**verbose** Prints shell input lines as they are read, but before any substitution has taken place. The lines are sent to stderr with a "+" at the beginning. The same as `set -v`. This is useful for debugging  
**vi** Sets inline editor to vi.  
**viraw** Process each character as it is typed in vi.  
**xtrace** Turns on trace mode. The same as `set -x`. This is useful for debugging. It forces the shell to echo out every command it is executing after it has done variable substitution *etc* but before it actually runs each command. The command lines are sent to stderr with a "+" at the beginning.

## Time based script execution

### Cron Table

Any user can create a cron table using the `crontab -e` command. But system administrators can control which users are allowed to create and edit cron tables with the `cron.allow` and `cron.deny`. To list the contents of the current user's cron table issue the `crontab -l`. For example:

```
15 3 8 1 * /usr/bin/somescript.ksh 2>&1 script.log
```

Will execute `somescript.ksh` on Jan 8 at 03:15  
Where \* is a weekday (0-6) 0 for Sunday through to 6 for Saturday.

If no log file is set the stdout and stderr will be emailed to the user.

### at command

We can use the `at` command to schedule a job to run at a specific time. System administrators can control which users can schedule jobs with the `at.allow` and `at.deny` files.

## I/O Redirection and Pipes

silent running: `myscript.ksh 2>&1 /dev/null`

**> file** Redirects standard output, **stdout**, to *file*.  
**< file** Redirects standard input, **stdin**, from *file*.  
**>> file** Appends redirected standard output, **stdout**, to *file*.  
**<< file** Opens *file* for reading and writing as standard input, **stdin**.  
**<< [-] word** Here document. Accepts shell input up to an occurrence of **word** on a line by itself (with no leading spaces or tab stops). the presence of the optional **"-"** causes all leading spaces or tab stops in the text to be stripped. Using *'word'* causes the text to be read uninterpreted (that is, **\$var** would be copied as **\$var** and not the value of **var**.)  
**n> file** *file* is the output for file designator *n*.  
0 is standard input, **stdin**.  
1 is standard input, **stdout**.  
2 is standard input, **stderr**.  
**n>& m** Redirects file designator *n* output to designator *m*.  
**<&** Closes standard input, **stdin**.  
**>&** Closes standard input, **stdout**.  
**<&p** Moves the input from the coprocess to the standard input, **stdin**.  
**>&p** Moves the input from the coprocess to the standard output, **stdout**.  
**1>&2** Merges standard output **stdout** into standard error, **stderr**.  
**cmd1|cmd2** Pipes *cmd1* standard output **stdout**, into *cmd2* standard input, **stdin**.

## Miscellaneous built-in commands

**alias [-tx] [name[=value]]**  
**alias** with no arguments prints the aliases. With arguments sets name to the same as value.  
**echo [arg]**  
Echoes *arg* to **stdout**.  
**eval [arg]**  
Evaluates the arguments as if the user had typed them in as text. **eval `command`** says to execute the output of **command** as if this had been typed into **stdin**.  
**exec [arg]**  
If *arg* is present, executes *arg* in place of this shell. (*arg* will replace this shell).  
**exit [n]**  
Terminates shell. The default is a return code of 0, otherwise **n** can be set from 0 through to 255 to give a specific return code.

**export [name[=value]]**  
Specifies the list of names to be exported to the environment of child processes.

**getopts opts\_string [arg]**  
Shell method for getting command-line arguments into a shell script. **!!!Expand!!!**

**kill**  
Kills a process

**let arg ((arg))**  
Korn shell expression capability. Arithmetic evaluation is done in the same manner as the **expr**

**print [-Rnprsu[n]] [arg]**  
Shell output mechanism. Prints text as specified. This command replaces **echo**.

**read [-prsu[n]] options**  
Shell input mechanism. Reads text as specified.

**readonly [name[=value]]**  
Sets specified variables once at read time; cannot be changed.

**return [n]**  
Immediately terminates the function. Default is a 0 return code; otherwise, **n** can be set from 0 through to 255. POSIX uses a range of 0 through to 125.

**set [+options] arg**  
Sets options for the shell. Most options correspond to shell flags. 1) No args 2) Reassign positional parameters. 3) set-o (see section on "set -o options")

**shift [n]**  
?????

**sleep time**  
Time for shell script to be inactive in seconds.

**ssh**  
?????

**test expression**  
Tests expression and gives a return code of in the \$? variable. A return code of 0 is true anything else is false.

**times**  
Prints the accumulated system and user times for this shell and child processes.

**time**  
?????

**timex**  
?????

**trap args sigs**  
Breaks execution in the shell and executes *args*

**typeset args [name[=value]]**  
Set the attributes and values for shell parameters and variables.

**ulimit [-afu] [n]**  
Imposes a size limit of *n* blocks. **-f** specifies that *n* equals 4096 blocks for each child process. **-a** displays current settings. **-u** ?

**umask [mask]**  
User file creation mask. The default file permissions. Common masks are 077, 027, 022.

**unalias name**  
Deletes name as an alias.

**unset [-f] name**  
Deletes name as an alias.

**whence [-pv] name**  
For each *name* specified, identifies how it will be used. **-v** produces a verbose output. **-p** does a path search for *name*.

**wait [PID]**  
Wait until child process has finished. If no option is set the script will wait until all child processes have finished.

## Upper or Lower Case Text for easy Testing

### Uppercasing

**uppercasevar=\$(echo \$variable | tr ' [a-z]' ' [A-Z]')**  
or  
**typeset -u variable**

### Downcasing

**downcasevar=\$(echo \$variable | tr ' [A-Z]' ' [a-z]')**  
or  
**typeset -l variable**

## Mail Notification Techniques

This is useful for notifying users when errors occur or a task has finished.  
**mail -s "The Subject" \$MAILOUT\_LIST < \$MAIL\_FILE**

---

This card was created using  $\LaTeX$ . The card may be freely distributed under the terms of the GNU general public license.  
Revision: 0.872, Date: 19/01/2009  
To contact me or download the latest version of this sheet please follow the links from: <http://www.BenjaminEvans.net>