How to make bash scripts work in dash

This page is an attempt to list some of the most common bashisms, i.e. features not defined by POSIX (won't work in dash, or general `/bin/sh`). It probably won't be exhaustive. Note also we talk about "bashism" because this wiki is largely bash-centric but a number (almost all) of these extensions work in at least some other shells like ksh or zsh with perhaps some differences in the details, as most of Bash's scripting features are derived from ksh. POSIX has simply required a much smaller number of them.

Syntax

	Works in bash	Change to for dash	Comment
defining functions	function f { echo hello world; }	f() { echo hello world; }	"function" is not defined by POSIX, only "name ()" is. The `function f {...}` syntax originated in `ksh` (and predates the Bourne syntax). In ksh both forms are present, but in the AT&T implementations, functions defined with "function" work slightly differently. `zsh` also supports both syntax without distinction.
case	`;;&` `;&` etc	None. Duplicate the case (use a function to avoid code duplication, or an alias to let the shell expand the whole segment as it parses the script)	`;;&` `;&` in bash4 is not defined by POSIX. AT&T ksh (since ksh88e, where it originated), MirBSD ksh (since R40) and zsh (since 3.1.2) have `;&` but not `;;&`

numeric C-like for loop	`for ((i=0; i<3; i++)); do`
	` echo "$i"`
	`done`
		`i=0 ; while [ "$i" -lt 3 ]; do`
		` echo "$i" ; i=$((i+1))`
		`done`
			this syntax is not defined by POSIX. Present in ksh93 where it originated and zsh.

expand sequences	`echo $'hello\tworld'`	`printf "hello\tworld\n"`	Historically `$' '` was not defined by POSIX through 2008, but has been accepted for the next version. http://austingroupbugs.net/view.php?id=249. Originated in ksh93, also supported by zsh
extended glob	`+( ) @( ) !( ) *( )`	not always possible, sometimes you can use several globs, sometimes you can use find(1)	not defined by POSIX. Originated in ksh. Supported by zsh with an option like bash.
select	`select`	some ideas: implement the menu yourself, use a command like dialog	not defined by POSIX. Originated in ksh, present in zsh.
file slurp	`$(< file)`	`$(cat file)`	Or read the file line by line.

Expansions

- Brace Expansion, eg `{a,b,c}` or `{1..10}` is not defined by POSIX. Both forms are present in zsh, ksh93, and the first form in older ksh and mksh. The first one originated in csh, the second in zsh.
- The `<( )` and `>( )` process substitutions are not defined by POSIX, but can be simulated with FIFOs: instead of `foo <(bar)`, write `mkfifo /tmp/foo_fifo; bar > /tmp/foo_fifo & foo /tmp/foo_fifo` (this is basically how process substitution is implemented on OSes that don't have a mechanism like `/dev/fd/` to refer to unnamed pipes with filenames). Originated in ksh93, also present in zsh.

Parameter Expansions

List of expansions not defined by POSIX:

- `${name:n:l}` -- You can use `$(expr "x$name" : "x.\{,$n\}\(.\{,$l\}\)")`. This originated in ksh93 and is also present in zsh.
- `${name/foo/bar}` -- you can use `$(printf '%s\n' "$name" | sed 's/foo/bar/')`, after changing shell patterns to regular expressions. This originated in ksh93 and is also present in mksh, and zsh, but ksh93's substitution expansion differs from Bash's.
- `${!name}` -- bash-specific; it is possible to use `eval` to achieve similar effects, but it requires great attention to detail; see BashFAQ/006.
- The behavior of the `#`, `##`, `%`, and `%%` operators are unspecified by POSIX and the ksh88 manual when used together with the `@` or `*` parameters. Dash applies the trimming to the flattened result. mksh/pdksh treats it as a bad expansion.

Note that using `$( )` has the side-effect of removing trailing newlines from the results. See CommandSubstitution for workarounds.

Arrays

Arrays are not defined by POSIX (but are present in ksh); there is no easy general workaround for arrays. Here are some hints:

- The positional parameters are a kind of array (only one array):

```
# Build a command dynamically. See BashFAQ/050
set -- 'mycommand' 'needs some complex' 'args'
"$@"
#access the i'th param
set -- one two three
i=2
eval "var=\${$i}" # i should be controlled by the script at all times. If influenced by side-effects
like user input, robust validation is required.
printf '%s\n' "$var"
```

- use `IFS` and `set -f`
- `eval` is powerful but easy to misuse in dangerous ways. See Eval command and security issues.

Conditionals

	Works in bash	Change to for dash	Comment
simple test	`[[`	use [and use double quotes around the expansions `[ "$var" = "" ]`	[[is not defined by POSIX, originated in ksh and is also present in zsh
pattern matching	`[[ foo = *glov ]]`	use `case` or `expr` or `grep`	see BashFAQ/041
equality with test	==	use = instead	only = is defined by POSIX
compare lexicographically.	`< >`	no change	present in dash, ksh, yash and zsh, but not defined by POSIX. See note below for possible workarounds.
compare modification times	`[[ file1 -nt file2 ]]` or `-ot`	`[ "$(find 'file1' -prune -newer 'file2')" ]` or `[ "file1" -nt "file2" ]`	`-prune` is required to avoid recursion; present in dash, ksh, yash and zsh. `-nt` and `-ot` aren't specified by POSIX.
check if 2 files are the same hardlink	`[[ file1 -ef file2 ]]`	`[ "file1" -ef "file2" ]`	`-ef` is not defined by POSIX, but is present in ksh, yash, zsh

			and Dash.
`(( ))`	`(( ))` (without the `$`) acts like a command on its own	For simple comparison: `[ -lt ] (and -ne -gt -ge)`. To assign a variable `var=$((3+1))`. For full functionality, use `[ "$(( (i+=2) < 5 && a > 3))" -ne 0 ]`.	present in ksh (where it originated) and zsh

Note: several standard POSIX utilities can be used for lexical comparisons. The examples below return a true (zero) exit status if the content of `$a` sorts before `$b`.

- `awk 'BEGIN { exit !(ARGV[1] "" < "" ARGV[2]) }' "$a" "$b"`
- `expr "x$a" "<" "x$b" >/dev/null`
- If the variables don't contain newline characters: `printf "%s\n" "x$a" "x$b" | sort -C` (also returns true if `$a` and `$b` are equal)

See http://austingroupbugs.net/view.php?id=375 for current work on extending the standard test builtin operators.

Arithmetic

See Arithmetic Precision and Operators and Arithmetic expansion for supported and required math expression features.

	Works in bash	Change to for dash	Comment
pre/post increment/decrement	`++ --`	`i=$((i+1))` or `: $((i+=1))`	
comma operator	`,`	`: "$((...))"; cmd "$((...))"`	The comma operator is *widely* supported by almost everything except dash and yash -- even posh and Busybox.
exponentiation	`**`		`**` is the only bash arithmetic operator that is not a standard C or C++ operator. ksh93 can use the standard `pow(x, y)` function, but since ksh93 is the only known shell to support `math.h` functions, it is not portable to POSIX shells in practice. The `**` operator is supported by at least bash, zsh, ksh93, and busybox, but not by dash or mksh.
	`let` or `((...))`	`[ "$((...))" -ne 0 ]`	Because of the above comma restriction, `let` can't be simulated exactly without a loop.

Redirections

	Works in bash	Change to for dash	Comment
redirect both stdout and stderr	`>&` and `&>`	`command > file 2>&1 or command 2>&1 | othercommand`	-
	`|&` (bash4)	`command 2>&1 | othercommand`	Conflicts with ksh. Not recommended, even in Bash. Just use `2>&1`.
duplicate and close	`m>&n- m<&n-`	`m>&n n>&-`	not defined by POSIX
herestring	`<<<"string"`	`echo | command`, or a here document to avoid a subshell (`<<EOF`)	-

Builtins

- `echo -n` or `-e` -- POSIX doesn't define any options, and furthermore allows `echo -e` to be the default behavior. Instead use `printf "%s\n"` (for normal echo) or `printf "%b\n"` (for `echo -e`); leave off the `\n` to simulate `echo -n`.
- `printf -v` is not defined by POSIX, and only Bash supports it. The `%q` and `%()T` formats are not defined by POSIX but supported by ksh93 and Bash. The a, A, e, E, f, F, g, and G formats are not required by POSIX for `printf(1)`, but dash appears to support `%f`, `%e`, `%E`, `%g`, and `%G`.
- `read` -- the only option defined by POSIX is `-r`; ksh has a different set of options that only partially overlaps with bash.
- `shopt`, and therefore all the options it provides (`extglob`, `nullglob`, `dotglob`, etc.) are not defined by POSIX and are bash-specific
- `local` -- there is no POSIX equivalent. You can use `$funcname_varname` to reduce the likelihood of conflicts, but even that is not enough for recursive functions. You can ensure that recursive calls occur in subshell environments (so there is a

"local" copy of all variables), or pass all "local variables" as parameters (because the positional parameters `$@`, `$1`, `$2`, etc are truly local). dash explicitly supports `local` as a non-Posix extension; ksh uses `typeset` instead, which works like bash's `declare`. `local` is mandated by the LSB and Debian policy specifications, though only the `local varname` (not `local var=value`) syntax is specified. An implementation of a variable stack for POSIX shells can be found there.

Special Variables

	Works in bash	Change to for dash	Comment
keep track of the times	`SECONDS`	`before=$(date +%s) ....seconds=$(( $(date +%s) - before))`	`date +%s` is not POSIX; see this faq for more info. Present in ksh
Generate a random number	`RANDOM`	`random=$(awk 'BEGIN{srand(); printf "%d\n", (rand()*256)}')` gives a number between 0 and 256 `random=$(hexdump -n 1 -e '/1 "%u"' /dev/urandom)` and `random=$(od -A n -N 1 -t u1 /dev/urandom)` give a timer-independent number between 0 and 256 `random=$(hexdump -n 2 -e '/2 "%u"' /dev/urandom)` and `random=$(od -A n -N 2 -t u2 /dev/urandom)` give a timer-independent number between 0 and 65535	Be sure to learn what `srand()` and `rand()` do, ie this method fails if you call `awk` several times rapidly. Instead generate all the numbers you need inside `awk`. Some systems also provide /dev/random and /dev/urandom , but this is not specified by the POSIX standard. ksh has `RANDOM`
Get the status of all the commands in a pipeline	`PIPESTATUS`	Simplest solution: `mkfifo fifo; command2 <fifo & command1 >fifo; echo "$?"` see NamedPipes	bash-specific; see this faq
Get the name of all / the current function name(s)	FUNCNAME	??	bash-specific see stackoverflow question

More

- The bash manual has a list of the differences between bash running in POSIX mode and a normal bash.
  Note: invoking bash in POSIX mode is only guaranteed to run a shell written according to the POSIX specification. It *doesn't* mean that it will fail if you use bashisms in your scripts.
- There is a handy perl script checkbashisms which is part of the debian devscripts package which can help point out bashisms in a particular script.
- https://wiki.ubuntu.com/DashAsBinSh The Ubuntu wiki also has a page that describes the differences
- Rich's sh tricks has some clever yet surprisingly powerful hacks for dealing with the limitations of the POSIX shell.

CategoryShell

Bashism (zuletzt geändert am 2016-12-01 21:39:00 durch GreyCat)