

Portability talk

The script programming language of BASH is based on the Bourne Shell syntax, with some extensions and derivations.

If scripts need to be portable, some of the BASH-specific syntax elements should be avoided. Others should be avoided for all scripts, e.g. if there is a corresponding POSIX®-compatible syntax (see Obsolete and deprecated syntax).


Some syntax elements have a BASH-specific, and a portable¹⁾ pendant. In these cases the portable syntax should be preferred.

construct	portable equivalent	Description	Portability
source FILE	. FILE	include a script file	Bourne shell (bash, ksh, POSIX®, zsh, ...)
declare keyword	typeset keyword	define local variables (or variables with special attributes)	ksh, zsh, ..., not POSIX!
command <<< WORD	command <<MARKER WORD MARKER	a here-string, a special form of the here-document, avoid it in portable scripts!	POSIX®
export VAR=VALUE	VAR=VALUE export VAR	Though POSIX® allows it, some shells don't want the assignment and the exporting in one command	POSIX®, zsh, ksh, ...

construct	portable equivalent	Description	Portability
<code>((MATH))</code>	<code>: \$((MATH))</code>	POSIX® doesn't define an arithmetic compound command, many shells don't know it. Using the pseudo-command <code>:</code> and the arithmetic expansion <code>\$(())</code> is a kind of workaround here. Attention: Not all shell support assignment like <code>\$((a = 1 + 1)) !</code> Also see below for a probably more portable solution.	all POSIX® compatible shells
<code>[[EXPRESSION]]</code>	<code>[EXPRESSION]</code> or <code>test EXPRESSION</code>	The Bashish test keyword is reserved by POSIX®, but not defined. Use the old fashioned way with the <code>test</code> command. See the classic test command	POSIX® and others
<code>COMMAND < <(... INPUTCOMMANDS...)</code>	<code>INPUTCOMMANDS > TEMPFILE</code> <code>COMMAND < TEMPFILE</code>	Process substitution (here used with redirection); use the old fashioned way (tempfiles)	POSIX® and others
<code>((echo X));(echo Y))</code>	<code>((echo X); (echo Y))</code>	Nested subshells (separate the inner <code>()</code> from the outer <code>()</code> by spaces, to not confuse the shell regarding arithmetic control operators)	POSIX® and others

Portability rationale

Here is some assorted portability information. Take it as a small guide to make your scripts a bit more portable. It's not complete (it never will be!) and it's not very detailed (e.g. you won't find information about how which shell technically forks off which subshell). It's just an assorted small set of portability guidelines. -*Thebonsai*

 **Fix Me!** UNIX shell gurus out there, please be patient with a newbie like me and give comments and hints instead of flames.

Environment (exported) variables

When a new value is assigned to an **existing environment variable**, there are two possibilities:

The *new value* is seen by subsequent programs

- without any special action (e.g. Bash)
- only after an explicit export with `export VARIABLE` (e.g. Sun's `/bin/sh`)

Since an extra `export` doesn't hurt, the safest and most portable way is to always (re-)export a changed variable if you want it to be seen by subsequent processes.

Arithmetics

Bash has a special compound command to do arithmetic without expansion. However, POSIX has no such command. In the table at the top, there's the `$((MATH))` construct mentioned as possible alternative. Regarding the exit code, a 100% equivalent construct would be:

```
# Bash (or others) compound command
if ((MATH)); then
...

# portable equivalent command
if [ "$((MATH))" -ne 0 ]; then
...
```

Quotes around the arithmetic expansion `$((MATH))` should not be necessary as per POSIX, but Bash and AT&T-KSH perform word-splitting on arithmetic expansions, so the most portable is *with quotes*.

echo command

The overall problem with `echo` is, that there are 2 (maybe more) mainstream flavours around. The only case where you may safely use an `echo` on all systems is: Echoing non-variable arguments that don't start with a `-` (dash) and don't contain a `\` (backslash).

Why? (list of known behaviours)

- may or may not automatically interpret backslash escape codes in the strings
- may or may not automatically interpret switches (like `-n`)
- may or may not ignore "end of options" tag (`--`)
- `echo -n` and `echo -e` are neither portable nor standard (**even within the same shell**, depending on the version or environment variables or the build options, especially KSH93 and Bash)

For these, and possibly other, reasons, POSIX (SUS) standardized the existence of the "printf" command.

Parameter expansions

- `${var:x:x}` is KSH93/Bash specific
- `${var/.../...}` and `${var//.../...}` are KSH93/Bash specific
- `var=$*` and `var=$@` are not handled the same in all shells if the first char of IFS is not " " (space). `var="$*"` should work (except the Bourne shell always joins the expansions with space)

Special variables

PWD

PWD is POSIX but not Bourne. Most shells are *not POSIX* in that they don't ignore the value of the `PWD` environment variable. Workaround to fix the value of `PWD` at the start of your script:

```
pwd -P > dev/null
```

RANDOM

RANDOM is Bash/KSH/ZSH specific variable that will give you a random number up to 32767 ($2^{15}-1$). Among many other available external options, you can use `awk` to generate a random number. There are multiple implementations of `awk` and which version your system uses will depend. Most modern systems will call 'gawk' (i.e. GNU `awk`) or 'nawk'. 'oawk' (i.e. Original/Old `awk`) does not have the `rand()` or `srand()` functions, so is best avoided.

```
# 'gawk' can produce random numbers using srand(). In this example,
10 integers between 1 and 500:
randpm=$(gawk -v min=1 -v max=500 -v nNum=10 'BEGIN { srand(systime()
+ PROCINFO["pid"]); for (i = 0; i < nNum; ++i) {print int(min + rand
() * (max - min))} }')
```

```
# 'nawk' and 'mawk' does the same, but needs a seed to be provided fo
r its rand() function. In this example we use $(date)
randpm=$(mawk -v min=1 -v max=500 -v nNum=10 -v seed="$(date +%Y%M%d%H%M%S)" 'BEGIN { srand(seed); for (i = 0; i < nNum; ++i) {print int(m
in + rand() * (max - min))} }')
```

Yes, I'm not an `awk` expert, so please correct it, rather than complaining about possible stupid code!

```
# Well, seeing how this //is// BASH-hackers.org I kinda missed the bash way of doing the above ;-)
# print a number between 0 and 500 :-)
printf $(( 500 * RANDOM / 32767 ))

# Or print 30 random numbers between 0 and 10 ;)
X=0; while (( X++ < 30 )); do echo $(( 10 * RANDOM / 32767 ));
done
```

SECONDS

SECONDS is KSH/ZSH/Bash specific. Avoid it. Find another method.

Check for a command in PATH

The PATH variable is a colon-delimited list of directory names, so it's basically possible to run a loop and check every PATH component for the command you're looking for and for executability.

However, this method doesn't look nice. There are other ways of doing this, using commands that are *not directly* related to this task.

hash

The hash command is used to make the shell store the full pathname of a command in a lookup-table (to avoid re-scanning the PATH on every command execution attempt). Since it has to do a PATH search, it can be used for this check.

For example, to check if the command ls is available in a location accessible by PATH :

```
if hash ls >/dev/null 2>&1; then
    echo "ls is available"
fi
```

Somewhat of a mass-check:

```
for name in ls grep sed awk; do
    if ! hash "$name" >/dev/null 2>&1; then
        echo "FAIL: Missing command '$name'"
        exit 1
    fi
done
```

Here (bash 3), hash also respects builtin commands. I don't know if this works everywhere, but it seems logical.

command



The command command is used to explicitly call an external command, rather than a builtin with the same name. For exactly this reason, it has to do a PATH search, and can be used for this check.

For example, to check if the command sed is available in a location accessible by PATH :

```
if command -v sed >/dev/null 2>&1; then
    echo "sed is available"
fi
```

¹⁾ "portable" doesn't necessarily mean it's POSIX, it can also mean it's "widely used and accepted", and thus maybe more portable than POSIX®

Discussion

 [scripting/nonportable.txt](#)  Last modified: 2019/08/30 16:30 by ersen

This site is supported by Performing Databases - your experts for database administration

Bash Hackers Wiki



Except where otherwise noted, content on this wiki is licensed under the following license:
GNU Free Documentation License 1.3