

Coding Standard for Bash Scripts

Perforce Professional Services

Version v2026.1, 2026-09-03

Table of Contents

Preface	1
1. Bash Version	2
2. Bash Directives	3
3. Scripts and Libraries	4
4. Script Templates	5
5. Script Sections	6
5.1. Header	6
5.2. Declarations and Environment	6
5.3. Local Functions	6
5.4. SDP Library Functions	6
5.5. Command Line Processing	7
5.6. Command Line Verification	7
5.7. Main Program	7
6. SDP Root and Relocatability	8
7. Version Identification	9
7.1. Short Form (Sourced Library Files)	9
7.2. Displaying Library Versions: <code>show_versions()</code>	10
7.3. Exemptions	10
7.4. Location	10
8. Logging	12
8.1. Log File Location and Naming	12
8.2. Log Symlink (LogLink)	12
8.3. Log Redirection	13
8.4. Controlling the Log	13
9. Standards	14
9.1. Tenets of Scripting	14
9.1.1. Avoid Requiring Modification	14
9.1.2. Self Logging	14
9.1.3. Error Handling	14
9.2. Style	15
9.2.1. Indentation	15
9.2.2. Naming Conventions	15
9.2.2.1. Shell Environment Variables	15
9.2.2.2. Global Script Variables	15
9.2.2.3. Library Output Variables	15
9.2.2.4. Summary of Naming Conventions	15
9.2.2.5. Function-scoped (Local) Variables	16
9.2.2.6. Function Names	16

9.2.3. Quoting	16
9.2.4. Conditionals	16
9.2.5. Line Length	16
9.2.6. Continuation Lines	17
9.2.7. Comments	17
9.3. Syntax Checking	17
9.4. ShellCheck Compliance	17
Appendix A: ShellCheck Notes	18
A.1. Lexicographic String Comparison in <code>[[]]</code>	18

Preface

Welcome to the Perforce P4 Server Deployment Package (SDP) Coding Standard for Bash scripts.

This Standard is intended to provide information useful for Bash script programming in the SDP Environment. This applies to scripts delivered as part of the SDP Package, and can also be applied to custom scripts such as custom triggers or systems integrations for interaction with P4 Servers that are added to the SDP in any given environment in [The Site Directory](#).

This document provides both *standards* and *guidelines* to follow. Standards must be followed for a script to be considered adherent to this standard. Guidelines are suggestions but are not strictly required to achieve compliance. Generally descriptions involving clear words like *must* indicate standards, while terms like *should* that allow for variation indicate guidelines.

Please Give Us Feedback

Perforce welcomes feedback. Please send suggestions for improving this document or the SDP to consulting-p4@perforce.com.

Chapter 1. Bash Version

SDP requires bash 4.0 or later. This is the floor needed for language features already relied on throughout the SDP, such as associative arrays (`declare -A`) and case-conversion parameter expansion (e.g. `${var,,}` to return the lowercase form of the value of `$var`), both introduced in bash 4.0.

The *shebang* line (the first line of each bash script that starts with the `#!` sequence) must be one of these two options:

```
#!/bin/bash
```

or

```
#!/usr/bin/env bash
```

Bash 4.0+ has been reliably available for a long time on Linux distributions that are not End of Life (EOL): Red Hat family distributions since version 7 (bash 4.2), Ubuntu since 10.04, and SUSE Enterprise Linux since 12.

As a whole, the SDP bash scripts support any UNIX/Linux environment that provides bash 4.0 or later. This includes all current Linux and most UNIX environments, except macOS (whose default system bash is 3.2; the workaround is installing a modern bash, e.g. via Homebrew, and adjusting the shebang line accordingly).

This standard applies to SDP on UNIX/Linux. For SDP on Windows, PowerShell, Python and Batch (`.bat`) scripts are used, so this standard for Bash does not apply. (Bash can run on Windows in various ways and some scripts may work, but are not supported due to not being tested.)

Chapter 2. Bash Directives

Immediately after the shebang line should appear the `set -u` directive, requiring variables to be defined before being referenced. The `set +u` directive is allowed as needed for examples, such as in command line processing where variables like `$1` may legitimately be referenced while undefined.

SDP scripts do not use `set -e`. Explicit error handling at each point where errors can occur is preferred instead. See the Error Handling tenet in the Standards section.

Chapter 3. Scripts and Libraries

A *script* is a bash shell script file intended to be executed directly by users or other automation, and for which the first line is the shebang line. Scripts have the `+x` execute bit set.

A *library* is a bash shell file intended to be sourced by scripts or other library functions. Library files have a `.lib` suffix, and do not have the `+x` execute bit set. Library files generally define reusable functions and may contribute to the shell environment.

Scripts appear in various directories in the deployed SDP structure.

Libraries should appear only in the `<SDPRoot>/common/lib` directory for scripts that are part of the SDP package, otherwise `<SDPRoot>/common/site/lib` for site-specific libraries.

Legacy exceptions: The `backup_functions.sh` remains in the `<SDPRoot>/common/bin` directory for backward compatibility with customer-side custom scripts. Some older library files may retain the `.sh` rather than `.lib` suffix.

Chapter 4. Script Templates

The bash script template illustrates and adheres to this standard. Here is the link to [the bash shell script template](#).

Chapter 5. Script Sections

Scripts should have whichever of the following sections apply:

5.1. Header

The Header section contains the shebang line, directives, license info, and indication of how to get documentation for the script.

Section Example - Header

```
#!/bin/bash
set -u

#=====
# Copyright and license info is available in the LICENSE file included with
# the Server Deployment Package (SDP), and also available online:
# https://workshop.perforce.com/view/p4-sdp/main/LICENSE
#=====
```

5.2. Declarations and Environment

This section declares global variables. Creating the shell environment for the script also starts in this section.

Section Example - Declarations and Environment

```
#=====
# Declarations and Environment

declare -i ErrorCount=0
```

5.3. Local Functions

Each script should define a subset of these standard local functions:

Section Example - Local Functions

```
#=====
# Local Functions

function msg () { echo -e "$*"; }
```

5.4. SDP Library Functions

Most scripts will use SDP libraries. Each library is sourced using the `$SDPCommonLib` variable.

Section Example - SDP Library Functions

```
#=====
# Load SDP Library Functions.

if [[ -d "$SDPCommonLib" ]]; then
    # shellcheck disable=SC1090 disable=SC1091
    source "$SDPCommonLib/logging.lib" ||\
        bail "Failed to load bash lib [$SDPCommonLib/logging.lib]. Aborting."
    # shellcheck disable=SC1090 disable=SC1091
    source "$SDPCommonLib/run.lib" ||\
        bail "Failed to load bash lib [$SDPCommonLib/run.lib]. Aborting."
fi
```

5.5. Command Line Processing

See the script template for an example of the Command Line Processing block.

5.6. Command Line Verification

See the script template for an example of the Command Line Verification block.

5.7. Main Program

See the script template for an example of the Main Program block. Among other things, this section is responsible for starting any log file processing that is to be done.

Chapter 6. SDP Root and Relocatability

In Production, the SDP Root Directory (referenced as the `$SDP_ROOT` shell environment variable or the `$SDPRoot` variable in scripts) will always have a value of `/p4`. However, the standard allows this root to be changed to simplify testing and development. All scripts should behave properly if this value is changed. In any production environment, the default value should apply; a value for `SDP_ROOT` should only be set for dev/test environments.

Chapter 7. Version Identification

All executable shell scripts must include a *Version ID Block*. Sourced library files (**.lib*) use a different, comment-only form — see below.

The regular form of the version ID block, for executable scripts, looks like this:

Version ID Block (Regular)

```
# Version ID Block. Relies on +k filetype modifier.
#-----
# shellcheck disable=SC2016
declare VersionID='$Id: //p4-
sdp/dev_rebrand/Server/Unix/p4/common/sdp_upgrade/sdp_upgrade.sh#6 $ $Change: 31803 $'
declare VersionStream=${VersionID#*///}; VersionStream=${VersionStream#*/};
VersionStream=${VersionStream%/*};
declare VersionCL=${VersionID##*: }; VersionCL=${VersionCL%% *}
declare Version=${VersionStream}.${VersionCL}
[[ "$VersionStream" == r* ]] || Version="${Version^^}"
```

This Version ID Block has several features:

- It ensures file versions are updated reliably on every submit by taking advantage of the P4 *+k* file type modifier to expand keywords in the file upon submit.
- The SDP major version is determined from the stream name (e.g., r25.2), so the version clarifies what released SDP version the file is part of.
- Non-released "dev branch" versions will have a version identifier that clearly indicates they are not released production code.
- The changelist number gives each file a unique identifier.

7.1. Short Form (Sourced Library Files)

Sourced *.lib* files must NOT use the regular, executable form of the block above. A library file is sourced into its caller's own shell process, so a `declare VersionID=/declare Version=` (etc.) in the library would silently overwrite the **calling script's* own same-named variables — confirmed behavior, since sourced code shares the caller's variable scope. A script that reads `$Version` after sourcing a library that also declared `$Version` would silently display the library's version instead of its own.

Instead, library files use a comment-only marker — never a *declare*, so nothing is ever actually assigned:

*Version ID Block (Short Form, for *.lib files)*

```
# Version ID Block (comment-only -- NOT executable). Relies on +k filetype
# modifier. This is a sourced library; a real 'declare' here would clobber
# the sourcing script's own $Version (confirmed: they share global scope).
# show_versions() greps this line's text directly; it is never evaluated.
```

```
# VersionID='$Id: //p4-sdp/dev_rebrand/Server/Unix/p4/common/lib/run.lib#4 $ $Change:
33368 $'
```

This is sufficient because nothing needs to evaluate a library's own `$Version` at runtime — `show_versions()` (see below) reads the `VersionID=` text directly out of the file with `grep`, whether it's inside a `#` comment or a real `declare` statement, and derives the same `Version` string via the same text processing the executable form's own derivation lines perform. The derivation lines themselves (`VersionStream=`, `VersionCL=`, `Version=`) are omitted from the library form since they'd never be evaluated anyway.

7.2. Displaying Library Versions: `show_versions()`

`utils.lib` provides a `show_versions()` function; any script that supports `-V/--version` and sources one or more `.lib` files should also source `utils.lib` in the same block (even if it doesn't otherwise need `utils.lib`), and wire its `-V/--version` option to call `show_versions()`. It prints the calling script's own version, plus the version of every `.lib` file the script actually sources — auto-detected by grepping the running script's own `source/.` lines, not by a manually-maintained list, so it can't drift out of sync as a script's sourcing changes.



A separate, older library mechanism (`libcore.sh/p4u_env.sh/libp4u.sh`, wired up via a script-maintained `$BASH_LIBS` list) still exists and is still used by a handful of scripts (`mkrep.sh`, `clear_depot_Map_fields.sh`). It predates the `*.lib` convention described in this document and is not compatible with it — the two are parallel, independent mechanisms. This is known technical debt to be consolidated later; for now, don't extend the older mechanism to new scripts, and be aware a script may be using either one when troubleshooting.

7.3. Exemptions

Some scripts are exempt from requiring a Version ID Block:

- Small, narrowly-focused wrapper scripts: `p4`, `p4d`, `p4p`, `p4broker`, `p4master_run`, `p4ftpd_base`, `p4d_1`, `ec2id`. These are too small/focused to be worth the overhead.
- `p4pstate.sh`, `p4dstate.sh`, `p4brokerstate.sh`: these are due for a broader overhaul (SDP-1050) and are exempted from versioning until that happens, rather than doing it twice.
- Test support scripts and other tiny/internal-only utility scripts (e.g. `get_mac_addresses.sh`).
- As of 2026.1, Python and Perl scripts are exempt as well — the Version ID Block mechanism above is bash-specific (it relies on bash parameter expansion to parse the `+k`-expanded `VersionID` string). Extending equivalent version identification to the SDP's Python and Perl scripts is tracked as a separate JIRA issue for a future release.

7.4. Location

Supported SDP scripts provided by Perforce appear in `/p4/common/bin`.

Custom scripts (inherently unsupported) are expected to appear somewhere under the

`/p4/common/site` directory, such as `/p4/common/site/bin` or `/p4/common/site/hms`.

Chapter 8. Logging

Scripts must be self-logging: all output generated during execution (stdout and stderr) is automatically captured in a log file without requiring the caller to use redirection or **tee**.

8.1. Log File Location and Naming

Log files are written to the **\$LOGS** directory, which is set by **p4_vars** to **/p4/<instance>/logs**. Each script run produces a time-stamped log file:

```
$LOGS/<ScriptName>.<YYYY-MM-DD-HHMMSS>.log
```

For example:

```
/p4/1/logs/daily_checkpoint.2026-04-07-143022.log
```

If two invocations start within the same second (e.g., a cron job and a manual run), an incrementing integer suffix is appended to guarantee uniqueness:

```
$LOGS/<ScriptName>.<YYYY-MM-DD-HHMMSS>.<N>.log
```

Log files are created atomically using the bash **noclobber** option (**set -C**), ensuring that concurrent invocations never claim the same filename.



Millisecond precision (**%3N**) is intentionally not used in fallback filenames because it is a GNU **date** extension not available on macOS. The integer suffix is sufficient to guarantee uniqueness and is fully portable.

8.2. Log Symlink (LogLink)

In addition to the time-stamped log file, each script maintains a stable symlink in the same directory:

```
$LOGS/<ScriptName>.log → $LOGS/<ScriptName>.<timestamp>.log
```

This **LogLink** symlink always points to the most recently started log, providing a predictable name for operators running **tail -f**, monitoring tools, and any integrations that need to locate the latest log without knowing the exact timestamp.

If a regular file already exists at the **LogLink** path (e.g., from an older SDP version that did not use symlinks), it is automatically renamed to a time-stamped filename before the symlink is created.

8.3. Log Redirection

After the log file and symlink are established, both stdout and stderr are redirected to the log via `exec`.

When running interactively (terminal attached, color mode active), a `tee` process substitution is used so that output appears on both the terminal and in the log simultaneously. ANSI color codes are stripped from the log copy so that log files remain clean when viewed with standard text tools.

In silent mode (`-si`), all output goes to the log only — nothing appears on the terminal. This is the intended mode for crontab invocations, where any terminal output would trigger an email from the cron daemon.

8.4. Controlling the Log

Two command-line options govern logging behavior:

`-L <file>`

Write the log to `<file>` instead of the default time-stamped file in `$LOGS`. No `LogLink` symlink is created in this case.

`-L off`

Disable logging entirely. All output goes to the terminal only. Cannot be combined with `-si`.

Chapter 9. Standards

9.1. Tenets of Scripting

9.1.1. Avoid Requiring Modification

No modification of scripts should be needed in customer environments for normal operation. An appropriate mix of command-line options and configuration files should be used to provide the flexibility required to operate to meet various customer needs.

9.1.2. Self Logging

Scripts must be self-logging: all output (stdout and stderr) is captured in a log file automatically, with no need for the caller to use redirection or `tee`. See [Chapter 8, Logging](#).

All scripts should support `-h` (short usage synopsis), `-man` (full documentation), and `-V` (version check, with `--version` alias) options, with standard meanings. All scripts must define a `usage()` function per the template. All scripts must have a `terminate()` function available, normally obtained by sourcing `logging.lib` rather than defined locally in each script. Exception: a small number of foundational, deliberately self-contained bootstrap scripts that source no SDP libraries at all (e.g. `install_sdp.sh`, `mkdirs.sh`) define their own local `terminate()` instead. All scripts should have complete documentation.

9.1.3. Error Handling

SDP scripts do not use `set -e`. Instead, errors are handled explicitly at each point where they can occur, using one of three mechanisms:

`bail`

Fatal error. Prints a red error message, increments `ErrorCount`, and exits immediately. Use this when continuing is not meaningful.

```
some_command || bail "some_command failed; cannot continue."
```

`errmsg`

Non-fatal error. Prints a red error message and increments `ErrorCount`, but execution continues. Use this when the script should keep running and report a failure count at the end.

```
some_command || errmsg "some_command failed; continuing."
```

`warnmsg`

Warning. Prints a yellow warning message and increments `WarningCount`. Use this for conditions that are noteworthy but not errors.

```
[[ -n "$SomeOptionalVar" ]] || warnmsg "SomeOptionalVar is not set."
```

At the end of the script, `ErrorCount` and `WarningCount` are checked to produce a final summary message and to set the exit code. The script exits with a value of `$ErrorCount`, so non-fatal errors still result in a non-zero exit code. The `terminate()` function (from `logging.lib`) handles this final exit.

9.2. Style

9.2.1. Indentation

Scripts must use 3-space indentation. Tab characters must not appear in script or library files, except within here-documents where tab characters carry semantic meaning.

9.2.2. Naming Conventions

9.2.2.1. Shell Environment Variables

Shell environment variables defined outside scripts — set in the surrounding shell environment or in SDP environment files such as `p4_vars` — must be all uppercase with underscore word separators, following standard UNIX/POSIX conventions. Examples: `SDP_ROOT`, `P4PORT`, `LOGS`.

9.2.2.2. Global Script Variables

Variables with global scope (declared outside any function) must use UpperCamelCase (PascalCase), starting with an uppercase letter. Examples: `ThisScript`, `ErrorCount`, `LogTimestamp`.

Constants — variables assigned once and not expected to change during the lifetime of the script — may alternatively use all-uppercase naming with underscore separators. Examples: `H1`, `H2`, `GREEN`, `RESET`.

9.2.2.3. Library Output Variables

Variables that are written by library functions and read by calling scripts must use `ALL_UPPERCASE` with underscore separators. This signals that the variable crosses an ownership boundary: the calling script did not set it — the library did. Examples: `CMDLAST`, `CMDEXITCODE`, `RCMDLAST`, `RCMDEXITCODE`.

This mirrors the convention for shell environment variables (which are also set outside the script) and gives a script reader an immediate visual cue: an all-uppercase name not listed in the script's own Declarations section is a library output, not a local variable.

9.2.2.4. Summary of Naming Conventions

Scope / Origin	Convention	Examples
Shell environment (set outside the script)	<code>ALL_UPPERCASE</code>	<code>SDP_ROOT</code> , <code>P4PORT</code> , <code>LOGS</code>
Global script variable (set by the script)	<code>UpperCamelCase</code>	<code>ThisScript</code> , <code>ErrorCount</code> , <code>LogTimestamp</code>

Scope / Origin	Convention	Examples
Constant (set once, does not change)	ALL_UPPERCASE or UpperCamelCase	H1, H2, GREEN, RESET
Library output variable (set by a library, read by the script)	ALL_UPPERCASE	CMDLAST, CMDEXITCODE
Function-local variable	lowerCamelCase	cmd, honorNoOpFlag, cmdOut
Function name	lowercase or lower_with_underscores	msg, bail, get_old_log_timestamp

9.2.2.5. Function-scoped (Local) Variables

Variables declared inside functions must use the `local` keyword and must use lowerCamelCase, starting with a lowercase letter. Examples: `cmd`, `desc`, `honorNoOpFlag`, `cmdOut`.



`declare` inside a function is functionally equivalent to `local` in Bash, but `local` is preferred for function-scoped variables because it more clearly expresses intent.

9.2.2.6. Function Names

Function names must be all lowercase. Multi-word function names use underscore separators. Examples: `msg`, `errmsg`, `bail`, `get_old_log_timestamp`.

9.2.3. Quoting

All variable expansions must be double-quoted unless word splitting or glob expansion is intentionally required. Use `"$var"` rather than `$var`.

9.2.4. Conditionals

Use `[[ ]]` (double brackets) rather than `[ ]` (single brackets) for all conditional tests. Double brackets are a Bash built-in with cleaner behavior for string comparisons, pattern matching, and avoidance of word-splitting surprises.

```
# Correct
[[ -n "$MyVar" ]]
[[ "$Count" -gt 0 ]]

# Avoid
[ -n "$MyVar" ]
[ "$Count" -gt 0 ]
```

9.2.5. Line Length

Lines should not exceed 120 characters. Keeping lines to 80 characters is preferred where practical, especially for documentation-heavy comment blocks.

9.2.6. Continuation Lines

When a command spans multiple lines, use `\` for line continuation and indent the continuation line by 3 additional spaces relative to the opening of the command.

9.2.7. Comments

Use `#` comments to explain intent, especially for logic that is not immediately apparent from the code. Inline comments are separated from code by at least two spaces.

Section headings use a standard major divider:

```
#=====
# Section Name
```

Function headers and sub-section breaks use a minor divider:

```
#-----
# Function: function_name
#
# Short description of what the function does.
#
# Input:
# $1 - first_param: Description.
# $2 - second_param: Description.
#
# Returns: 0 on success, non-zero on error.
#-----
```

9.3. Syntax Checking

Every script and library file must pass `bash -n <file>` with no errors before being considered complete. This is a fast, mechanical syntax check, independent of and complementary to ShellCheck's more thorough static analysis below — run it first.

9.4. ShellCheck Compliance

The ShellCheck utility is a static code analysis tool for bash shell scripts. See: <https://www.shellcheck.net>

All SDP scripts and library files must pass a ShellCheck scan with ShellCheck version 0.10.0 or later.

Where appropriate, `#shellcheck disable=SC<NNNN>` directives may be used, as may `.shellcheckrc` files, to suppress warnings deemed not of concern. See [Appendix A, ShellCheck Notes](#) for cases where ShellCheck guidance conflicts with SDP style.

Appendix A: ShellCheck Notes

This appendix documents cases where ShellCheck guidance conflicts with SDP style, and the rationale for the SDP's position.

A.1. Lexicographic String Comparison in `[[ ]]`

SDP scripts sometimes need to compare two version-like strings lexicographically (e.g. comparing P4D version strings such as `2025.1` against `2026.1`) rather than numerically, since these values aren't valid integers to bash. The idiom used is:

```
# shellcheck disable=SC2072
[[ "$OldVersion" < "$NewVersion" ]]
```

ShellCheck's SC2072 warns that `</>` inside `[[ ]]` might be a mistake for the numeric comparison operators `-lt/-gt`, since that confusion is a common error. Here the string comparison is intentional, so the warning is suppressed with an explicit `# shellcheck disable=SC2072` comment immediately above the line — both to silence the warning and to signal to a future reader that the string comparison is deliberate, not a mistake.