How can I write a PRACTICE script that runs the same way on different host operating systems?

2026-10-21 - Comments (0) - Setup / update

Most portability comes from using TRACE32's own mechanisms instead of host-OS-specific ones, and from isolating the few things that genuinely differ.

The typical failure scenario is a script that was written and tested on Windows and then silently breaks on a Linux or macOS host: a path containing backslashes, a `dir` command passed to `OS.Command`, or a serial port name that only exists on one platform. Nearly all of these problems can be avoided by following a few simple rules. The underlying principle is: **prefer TRACE32's OS-neutral mechanisms, and confine OS-specific code to the smallest possible set of explicit, well-commented branches.**

Use OS-Neutral Paths

Do not hard-code absolute paths such as `C:\projects\...` or `/home/user/...`. TRACE32 provides two mechanisms that make paths portable.

1) Use forward slashes in all paths. TRACE32 converts `/` to the host's path separator at runtime, so a script containing `Data.LOAD.Elf "~~~~/myprog.elf"` works on every host, while the same path written with backslashes, `Data.LOAD.Elf "~~~~\myprog.elf"`, is only valid on Windows. This rule applies everywhere a path appears — command arguments, `OS.FILE()` checks, `COPY` sources, and so on.

2) Use path prefixes instead of hard-coded directories. The prefixes are resolved by TRACE32 against the current installation and user environment:

PREFIX	MEANING
`~/`	Home directory (from `$HOME`)
`~~/`	TRACE32 system directory
`~~~/`	Temporary directory of TRACE32
`~~~~/`	Directory of the running script
`./`, `../`	Current / parent directory

Using `~~~~/` is particularly valuable for scripts that load their own data files: the script keeps working no matter where the user places it, as long as the relative layout is preserved.

When you need a path as a string rather than as a literal — for concatenation, comparison, or building longer paths — use the dedicated PRACTICE functions: `OS.PresentPracticeDirectory()` returns the directory of the running script, `OS.PresentHomeDirectory()` the home directory, and `OS.FILE.JOINPATH()` joins path elements with the correct host separator. This is the right way to build dynamic paths such as a log file inside the script directory.

Let TRACE32 Do File Operations — Avoid the Host Shell

For everyday file work — copying, renaming, creating or removing directories, zipping archives — use TRACE32's own commands: `COPY`, `REN`, `MKDIR`, `RMDIR`, `ZIP`, and `RM`. TRACE32 translates these to the correct native calls for the running host, so the *same script line* behaves identically on Windows, Linux, and macOS. There is nothing to branch on.

The trap is the opposite: passing native shell commands through `OS.Command`, `OS.Area`, or `OS.screen`. A command like `dir` works on Windows and fails on Linux, where the equivalent is `ls`; file-copying, archiving, and permission commands differ similarly. In practice, `OS.Command` with a native shell command is the single most common cause of cross-platform script failures. Reserve `OS.Command` for things that genuinely require the host shell, and guard it behind an `OS.NAME()` check.

Detect the Host OS with OS.NAME()

Some differences are unavoidable — the most typical example is device naming, where a serial port is `COM1` on Windows but `/dev/ttyS0` on Linux. Where such a difference cannot be designed around, detect the host operating system with the PRACTICE function `OS.NAME()` and branch on it. `OS.NAME()` returns the name of the host operating system, e.g, *"Windows"*, *"Linux"* and *"MacOSX"*

A typical branch — opening a serial terminal with the port name that exists on the host:

```
LOCAL &os
&os = OS.NAME()
IF ("&os"=="Windows")
(
    TERM.METHOD COM COM1 9600. 8 NONE 1STOP NONE
)
ELSE
(
    TERM.METHOD COM /dev/ttyS0 9600. 8 NONE 1STOP NONE
)
```

Keep such branches minimal and clearly commented. Every branch you add is OS-specific code that must be maintained and re-tested on every platform, so treat `OS.NAME()` branches as a last resort: first ask whether an OS-neutral mechanism solves the problem.

Check File Existence and Access Before Use

Robust scripts check their preconditions instead of assuming files and permissions are there. The PRACTICE function `OS.FILE()` tests whether a file exists; the access PRACTICE functions `OS.ACCESS()`, `OS.FILE.ACCESS()`, and `OS.DIR.ACCESS()` test whether a file or directory has the required permissions — both work on Windows and Linux. The `OS.ACCESS()` documentation details how the access types (`R`, `W`, `X`, `D`, `K`) map to concrete permissions on each OS, so check which type you actually need rather than relying on a guess.

Checking first lets the script fall back gracefully — print a helpful message, use a default file, skip a step — instead of failing with an opaque error on a host where the file simply does not exist. The same idea applies to hardware: probe for debug hardware before using it, so the script degrades instead of failing on hosts without the device attached.

Know the Trap

- Accessing environment variables is a common portability trap. If your script needs to read or set environment variables, study the shipped example `~~/demo/practice/my_environment.cmm`, which shows how to handle this cross-platform. Where possible, avoid environment variables altogether and use the `~/` prefix or `OS.PresentHomeDirectory()` for home-directory paths.

- TRACE32 reads `LF` line endings on all hosts, so input files are safe. The asymmetry shows up when *writing* files: TRACE32 uses the host's native end-of-line sequence, so the same script produces CRLF files on Windows and LF files on Linux. If byte-identical output matters (e.g., for checksums or downstream parsers), write the line terminator explicitly (e.g., `%Ascii 0x0A`) and verify the output on each OS.

- The `WAIT` command can behave slightly differently depending on the host OS, so do not build timing-critical logic on it. Wherever possible, poll for the actual condition you are waiting for instead of assuming a fixed delay has passed; and if you do use `WAIT`-based sequences, test them on all target OSes.

Complete Example

A minimal cross-platform script combining the ideas above: resolve the script directory portably, load a data file if it exists (falling back to a wildcard otherwise), open the serial terminal with the host-appropriate port name, and back up a config file — all without a single hard-coded path or shell command.

```
LOCAL &ppd &elf

&ppd = OS.PresentPracticeDirectory()
&elf = "&ppd/out/myprog.elf"

IF OS.FILE("&elf") == TRUE()
    Data.LOAD.Elf "&elf"
ELSE
    Data.LOAD.Elf "*.elf"

IF OS.NAME()=="Windows"
    TERM.METHOD COM COM1 9600. 8 NONE 1STOP NONE
ELSE
    TERM.METHOD COM /dev/ttyS0 9600. 8 NONE 1STOP NONE
```

Rules of Thumb

When in doubt, run through this checklist before declaring a script portable:

1. Are all paths using forward slashes and/or the standard prefixes?
2. Are file operations done with `COPY`/`REN`/`MKDIR`/`RMDIR`/`ZIP`/`RM` instead of `OS.Command`?
3. Is every `OS.NAME()` branch justified, small, and commented?
4. Are files and hardware checked for existence before use?
5. Are environment variables, line endings, and `WAIT`-based timing reviewed for OS dependence?
6. Has the script actually been run on Windows, Linux, and macOS — not just on the author's machine?

Details on file handling are in the Training Script Language PRACTICE and the PowerView Command Reference; per-function reference in the PowerView Function Reference.