Technical Note TN2065
do shell script in AppleScript

CONTENTS

This Technote describes basic techniques and answers common questions for AppleScript's *do shell script* command, which was introduce in AppleScript 1.8.

[Jan 27 2003]

Issuing Commands

Q: My command works fine in Terminal, but when I try to use it in *do shell script*, I get an error about "command not found."
A: There are two possibilities: first, *do shell script* always uses the `sh` shell, not your default shell, which Terminal uses. (The default shell for new users is `csh`, so unless you've changed it, that's what it is.) While some commands are the same between shells, others are not, and you may have used one of them. If you write your *do shell script* scripts in Terminal first, always use `sh`. You can start `sh` by typing "`/bin/sh`"; type "`exit`" to get back to your normal shell.

Second, when you use just a command name, the shell uses a list of directories (known as your `PATH`) to try and find the complete path to the command. For security and portability reasons, *do shell script* uses its own list, which may not match the one your default shell uses. Use the full path to the command, e.g., `/sbin/ifconfig` instead of just `ifconfig`. To find the full path in Terminal, say "`which command-name`", e.g., "`which ifconfig`"; to see the list of places *do shell script* will search, say "*do shell script "echo $PATH"*".

Q: Why doesn't *do shell script* work exactly like Terminal?
A: For two reasons: first, it helps guarantee that scripts will run on different systems without modification. If *do shell script* used your default shell or `PATH`, your script would probably break if you gave it to someone else. Second, it matches shell escape mechanisms in other languages, such as Perl.

Q: How do I run my command with a shell other than `sh`?
A: Include the shell you want to use explicitly in the command. There are a variety of ways to pass commands to your shell of choice. You could write the command to a file and then execute the file like this:

```
do shell script "/bin/tcsh my-command-file-path"
```

Some shells will accept a script as a parameter, like this:

```
do shell script "/bin/tcsh -c 'my-command'"
```

And most will accept a script from standard input, like this:

```
do shell script "echo my-command | /bin/tcsh"
```

When in doubt, read the documentation for your preferred shell. When you put the command in the *do shell script* string, you will probably have to quote the command, or `sh` will interpret special characters in the command.

Q: How can I use more than one command in a single *do shell script*? For example, I want to `cd` to some directory and then do some work, but it doesn't remember the working directory from one invocation to the next.
A: Each invocation of *do shell script* uses a new shell process, so state such as changes to variables and the working directory is not saved from one to the next. To do several commands in a single invocation, separate the commands with

semicolons like this:

```
do shell script "cd ~/Documents; ls"
-- result: "Welcome.txt"
```

Due to a bug in how authentication works, this does not work correctly with *with administrator privileges* – see below.

Q: How do I get administrator privileges for a command?
A: Use the *administrator privileges* and *password* parameters like this:

```
do shell script "command" password "mypassword" with administrator privileges
```

If you omit the *password* parameter, *do shell script* will ask for a password when it runs.

Bear in mind that administrator privileges allow you to change any file anywhere in the system. You can render your system unbootable or even erase the entire disk with a few well-placed commands, so exercise caution. Better yet, don't use administrator privileges unless you absolutely have to. Unless you are doing system-level development, you should never need to change anything in `/System` – changing `/Library` should suffice.

Because of a bug, *administrator privileges* does not work correctly with multiple commands. You must turn your command into a single invocation of `sh`, like this:

```
set normal_command to "command1; command2"
do shell script "sh -c " & quoted form of normal_command with administrator privileges
```

Back to top

Getting an answer

Q: How does *do shell script* get the result?
A: Shell commands can write their results to one of two output streams: standard output and standard error. Standard output is for normal output, while standard error is for error messages and diagnostics. Assuming your script completes successfully – if it doesn't, see the next question – the result is whatever text was printed to standard output, possibly with some modifications.

By default, *do shell script* transforms all the line endings in the result to Mac-style carriage returns (*"\r"* or *ASCII character 13*), and removes a single trailing line ending, if one exists. This means, for example, that the result of "*do shell script "echo foo"*" is simply *"foo"*, not the *"foo\n"* that `echo` actually returned. You can suppress both of these behaviors by adding the *without altering line endings* parameter. For dealing with non-ASCII data, see Dealing With Text.

Q: How does *do shell script* report errors?
A: All shell commands return an integer status when they finish: zero means success; anything else means failure. If the script exits with a non-zero status, *do shell script* throws an AppleScript error with the status as the error number. (The man page for a command should tell you what status codes it can return. Most commands simply use 1 for all errors.) If the script printed something to the standard error stream, that text becomes the error message in AppleScript. If there was no error text, the normal output (if any) is used as the error message.

Q: When I run my command in Terminal, I get a bunch of output, but when using *do shell script*, some of it is missing.
A: When running in Terminal, standard output and standard error are both sent to the same place, so it's difficult to tell them apart. *do shell script*, on the other hand, keeps the two streams separate. If you want to combine them, follow the command with `2>&1` like this:

```
do shell script "command 2>&1"
```

For more details, see the `sh` man page under "Redirections."

Back to top

Dealing With Text

Q: My command doesn't work right when a parameter has spaces or certain punctuation – parentheses, $, *, etc.
A: Because the shell separates parameters with spaces, and some punctuation marks have special meanings, you must take special steps to make the shell treat your string as one parameter with literal spaces, parentheses, etc. This is called "quoting," and there are several ways to do it, but the simplest and most effective is to use the *quoted form* property of strings.

For example, consider this (buggy) handler, which takes a string and appends it to a file named "stuff" in your home directory:

```
to append_message(s)
    do shell script "echo " & s & " >> ~/stuff"
end append_message
```

It works fine for most strings, but if we call it with a string like "$100", the string that ends up in the file is "00", because the shell thinks that "$1" is a variable whose value is an empty string. (Variables in `sh` begin with a dollar sign.) To fix the script, change it like this:

```
do shell script "echo " & quoted form of s & " >> ~/stuff"
```

The *quoted form* property gives the string in a form that is safe from further interpretation by the shell, no matter what its contents are. For more details on quoting, see the `sh` man page under "Quoting."

Q: I need to put double quotes and backslashes in my shell command, but AppleScript gives me a syntax error when I try.
A: Strings in AppleScript go from an opening double quote to a closing double quote. To put a literal double quote in your string you must "escape" it with a backslash character, like this:

```
"a \"quote\" mark"
```

The backslash means "treat the next character specially." This means that getting a literal backslash requires two backslashes, like this:

```
"a back\\slash"
```

Putting this all together, you might have something like this:

```
set s to "this is a test."
do shell script "echo " & quoted form of s & " | perl -n -e 'print \"\\U$_\"'"
-- result: "THIS IS A TEST."
```

Despite all the extra backslashes in the script, the actual string passed to perl's -e option is

```
print "\U$_"
```

Q: Whenever my shell script returns a double quote or backslash, it comes out with an extra backslash in front of it.
A: The result window shows you the result in "source" form, such that you could paste it into a script and compile it. This means that string results have quotes around them, and special characters, such as double quotes and backslashes, are escaped as described above. The extra backslash is not really part of the string, it's merely how it's displayed. If you pass the string to *display dialog* or write it to a file, you'll see it without the extra backslashes.

Q: What does *do shell script* do with non-ASCII text (accented characters, Japanese, etc.)?
A: As of AppleScript 1.8.3, *do shell script* handles all its input and output as UTF-8. This works correctly with file names and as well as possible with the commands themselves. (Before that, it used the user's primary text encoding, which made it extremely difficult to deal with files that had non-ASCII characters in their names.)

There is currently no support for encodings other than UTF-8, and if a command produces non-ASCII characters that are not valid UTF-8 (e.g., `cat`-ing a text file saved with the MacRoman encoding) *do shell script* will return an error that it "can't make some data into the expected type." Workarounds include writing the output to a file and then reading it using AppleScript's *read* command or piping through `vis`.

Realize that most shell commands are completely ignorant of Unicode and UTF-8. UTF-8 looks like ASCII for ASCII characters – for example, "*A*" is the byte 0x41 in both ASCII and UTF-8 – but any non-ASCII character is represented as a sequence of bytes. As far as the shell commands are concerned, however, one byte equals one character, and they make no attempt to interpret anything outside the ASCII range. This means that they will preserve UTF-8 sequences and can do exact byte-for-byte matches: for example, `echo "™"` will produce a trademark symbol, and `grep "α"` will find every line with a lowercase alpha. However, they cannot intelligently sort, alter, or compare UTF-8 sequences: for example, character-set matching commands like `tr` or the `[]` construct in `sed` will attempt to match each byte of the sequence independently, `sort` will sort accented characters out of order, and `grep -i` or `find -iname` will not match "é" against "É". Perl is a notable exception to this mess, but you will probably need to add `use utf8` to your Perl script. See the `perlunicode` man page for more details.

Q: What are the rules for line endings?

A: There are two different line ending conventions in Mac OS X: Mac-style (lines end with return: *"\r"* or *ASCII character 13*) and Unix-style (lines end with line-feed: *"\n"* or *ASCII character 10*). Shell commands typically only handle Unix-style line endings, so giving them Mac-style text will produce less-than-useful results. For example, `grep` would consider the entire input to have only one line, so you would get at most one match.

If your data is coming from AppleScript, you can transform the line endings there or generate line feeds in the first place – *"\n"* or *ASCII character 10* both yield a line feed. If your data is coming from a file, you can make the shell script transform the line endings by using `tr`. For example, the following will find lines that contain "something" in any plain text file. (The "*quoted form of POSIX path of f*" idiom is discussed under Dealing With Files:)

```
set f to choose file
do shell script "tr '\\r' '\\n' < " & quoted form of POSIX path of f & " | grep
something"
```

AppleScript itself is line ending-agnostic – the *paragraph* element of *string* and *Unicode text* objects considers Mac, Unix, and Windows-style line endings to be equivalent. There is generally no need to use *text item delimiters* to get the lines of Unix-style text; *paragraph n* or *every paragraph* will work just as well. (However, if you wanted to consider *only* Unix-style line endings, *text item delimiters* would be the proper solution. Also, prior to AppleScript 1.9.1, *Unicode text* objects only considered return and the Unicode paragraph-separator character to be paragraph breaks.)

Back to top

Dealing With Files

Q: I've got an AppleScript *file* or *alias* object; how do I pass it to a shell command?
A: The shell specifies files using POSIX path names, which are strings with slashes separating the path components (e.g., `"/folder1/folder2/file"`). To get the POSIX path of an AppleScript *file* or *alias* object, use the *POSIX path* property. (However, see the following question.) For example:

```
POSIX path of file "HD:Users:me:Documents:Welcome.txt"
-- result: "/Users/me/Documents/Welcome.txt"
```

To go the other way – say your shell command returns a POSIX path as a result – use the *POSIX file* object. *POSIX file* with a path name evaluates to a normal *file* object that you can then pass to other AppleScript commands. For example:

```
set p to do shell script "echo ~"
POSIX file p
-- result: file "HD:Users:me:"
```

Q: *POSIX path* doesn't work right if the file has certain characters in its name – spaces, parentheses, $, *, etc.
A: This is a special case of quoting: you must quote the path to make the shell interpret all the punctuation literally. To do this, use the *quoted form* of the path. For example, this will work with any file, no matter what its name is:

```
choose file
do shell script "ls -l " & quoted form of the POSIX path of the result
-- result: "-rw-r--r-- 1 me unknown 1 Oct 25 17:48 Look! a file!"
```

Q: Why doesn't *POSIX path* just quote everything for me?
A: For two reasons: first, there are uses for POSIX paths that have nothing to do with shell parameters, and quoting the path would be wrong in such cases. Second, *quoted form* is useful for things other than file paths. Therefore, there are two orthogonal operations instead of one combined one.

Back to top

Other Concerns

Q: How do I control an interactive tool like `ftp` or `telnet` with *do shell script*?
A: The short answer is that you don't. *do shell script* is designed to start the command and then let it run with no interaction until it completes, much like the backquote operator in most Unix shells or the `system` call in awk and Perl.

However, there are ways around this. You can script Terminal and send a series of commands to the same window (though this only works in Mac OS X 10.2 and later), or you could use a Unix package designed for scripting interactive tools, such as `expect`. Also, many interactive commands have non-interactive equivalents. For example, `curl` can substitute for `ftp` in most cases.

Q: My script will produce output over a long time. How do I read the results as they come in?
A: Again, the short answer is that you don't – *do shell script* will not return until the command is done. What you can do,

however, is to put the command into the background (see the next question), send its output to a file, and then read the file as it fills up.

Q: I want to start a background server process; how do I make *do shell script* not wait until the command completes?
A: Use "*do shell script "command > file_path 2>&1 &*"". *do shell script* will return immediately with no result and your AppleScript script will be running in parallel with your shell script. The shell script's output will go into *file_path*; if you don't care about the output, use "`/dev/null`". There is no direct support for getting or manipulating the background process from AppleScript.

Q: I'm trying to use `top`, but it fails saying "can't get terminal attributes" or "error opening terminal: unknown."
A: `top` in its default mode does all sorts of clever things to create a dynamically updating display, none of which work if the output device does not support cursor control, as *do shell script* does not. However, `top` has an option that makes it run in logging mode, which works with file-like devices like *do shell script*. Use `top -l1` instead, or see the `top` man page for more options.

This same problem will apply to any other command that assumes the presence of a terminal. Fortunately, most of them are interactive front ends to more primitive commands that do not assume a terminal.

Q: What's the default working directory for *do shell script* commands?
A: *do shell script* inherits the working directory of its parent process. For most applications, such as Script Editor, this is the working directory of its parent process, the Finder, which is "/". For `osascript`, it's the working directory of the shell when you launched `osascript`. You should not rely on the default working directory being anything in particular. If you need the working directory to be someplace specific, set it to that yourself.

Back to top

Downloadables

 Acrobat version of this Note (152K) Download

Back to top
