Technical Note TN2065

do shell script in AppleScript

This Technote answers frequently asked questions about AppleScript's do shell script command, which was introduced in AppleScript 1.8.

This technical note addresses common questions about how to use do shell script. It does not attempt to explain what you can do with a Unix shell script or how to write one; for that, find an appropriate Unix text or consult a local expert. It is structured as question-and-answer, so you can either skip right to your problem, or read through from beginning to end.

Some answers refer to "man pages"; these are reference documents included in Mac OS X. (The "man" is short for "manual.") To see the man page for a command, open a Terminal window and type `man` followed by the command name, for example, `man echo`.

Issuing Commands

How do I pass an AppleScript variable to my shell command?

Since the command argument to do shell script is really just a string, you can build the string at run time using the AppleScript concatenation operator `&`. For example, to pass the variable as a command parameter, you would do this:

```
set hostname to "www.apple.com" do shell script "ping -c1 " & hostname
```

Some commands require data to be fed to standard input. do shell script does not directly support this, but you can fake it using `echo` and a pipe:

```
set input to "hello" do shell script "echo " & input & " | tr a-z A-Z" -- "HELLO"
```

In general, you should quote any variables using `quoted form of`; see Dealing with Text for details.

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." What's going on?

There are two possibilities. First, do shell script always uses /bin/sh to interpret your command, not

your default shell, which Terminal uses. (To find out what your default shell is, say `echo $SHELL` in Terminal.) 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 instead of a complete path, 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 ignores the configuration files that an interactive shell would read, so you don't get the customizations you would have in Terminal. Use the full path to the command, for example, `/sbin/ifconfig` instead of just `ifconfig`. To find the full path in Terminal, say `which command-name`, for example, `which ifconfig`; to see the list of places do shell script will search, say `do shell script \"echo $PATH\"`.

(This answer glosses over a few details — see Gory Details if you care.)

Why doesn't do shell script work exactly like Terminal?

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.

How do I run my command with a shell other than sh?

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 as described below, or sh will interpret special characters in the command.

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.

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"
```

Using a line feed (ASCII character 10) also works.

How do I get administrator privileges for a command?

Use the `administrator privileges`, `user name` and `password` parameters like this:

```
do shell script "command" user name "me" password "mypassword" with administrator
privileges
```

`user name` and `password` are optional; if you omit the user name, do shell script assumes it to be the current user; if you omit the password, it will ask for a password when it runs. Once a script is correctly authenticated, it will not ask for authentication again for five minutes. As of Mac OS X 10.4, this grace period does not extend to any other scripts or to the rest of the system; manually calling `sudo -k` is unnecessary.

For security reasons, you may not tell another application to do shell script with administrator privileges. Put the command outside of any tell block, or put it inside a tell me block.

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.

Note: Using sudo(8) with `with administrator privileges` is generally unnecessary and creates security holes; simply remove the "sudo".

Warning: In Mac OS X 10.4.0 and 10.4.1, `with administrator privileges` executes the command with only the effective user id set to root. This causes trouble for some commands that rely on the real user id — for example, Perl will turn on its "taint mode" security checks, and sudo(8) will hang. To work around the problem (assuming you cannot simply remove a use of sudo; see above), preface your command with a small Perl script to set the real user id, like this:

```
do shell script "/usr/bin/perl -Ue '$< = $>; system(@ARGV)' my_command" with
administrator privileges
```

Mac OS X 10.4.2 sets both the real and effective user ids; the workaround described here will be unnecessary, but harmless.

Warning: Prior to Mac OS X 10.4, `with administrator privileges` did not work correctly with multiple commands. You had to turn your multiple commands 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
```

As of Mac OS X 10.4, you can use `with administrator privileges` with multiple commands as described in "how can I use more than one command" above; no workaround is necessary.

How long can my command be? What's the maximum number of characters?

There is no precise answer to this question. (See Gory Details for the reasons why.) However, the approximate answer is that a single command can be up to about 262,000 characters long — technically, 262,000 bytes, assuming one byte per character. Non-ASCII characters will use at least two bytes per character — see Dealing with Text for more details.

Note: This limit used to be smaller; in Mac OS X 10.2 it was about 65,000 bytes. The shell command `sysctl kern.argmax` will give you the current limit in bytes.

Overrunning the limit will cause do shell script to return an error of type 255. Most people who hit the limit are trying to feed inline data to their command. Consider writing the data to a file and reading it from there instead.

Back to Top

Getting an Answer

How does do shell script get the result? How do I return a shell variable to my AppleScript?

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. Most commands print their results to standard output automatically, so you don't need to do anything extra. If your answer is in a variable, you will need to print it yourself using `echo` (most shells) or `print` (many languages, such as Perl and Awk). For example:

Listing 1: Set the AppleScript variable mySlug to a date slug in yyyy-mm-dd format.

```
set mySlug to do shell script "date +%Y-%m-%d" -- see the 'date' man page for details
on the format string.
```

Listing 2: The same thing, but as a Perl script.

```
set mySlug to do shell script ¬     "perl -e 'my (undef, undef, undef, $d, $m, $y) =
localtime;               my $date = sprintf("%4d-%02d-%02d", $y+1900, $m+1, $d);
print $date'"
```

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; echo bar\"` is `\"foo\\rbar\"`, not the `\"foo\\nbar\\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.

How does do shell script report errors?

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.

When I run my command in Terminal, I get a bunch of output, but when using do shell script, some of it is missing.

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"
```

See the sh man page under "Redirection" for more details.

How much output can a command return?

A single command can return up to 1GB of data.

Back to Top

Dealing with Text

My command doesn't work right when a parameter has spaces or certain punctuation — parentheses, $, *, etc.

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."

I need to put double quotes and backslashes in my shell command, but AppleScript gives me a syntax error when I try.

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$_"
```

Whenever my shell script returns a double quote or backslash, it comes out with an extra backslash in front of it.

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.

What does do shell script do with non-ASCII text (accented characters, Japanese, etc.)?

From AppleScript's point of view, do shell script accepts and produces Unicode text. do shell script passes the commands to the shell and interprets their output using UTF-8. If a command produces bytes that are not valid UTF-8, do shell script will interpret them using the primary system encoding.

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 \"©\"` will find every line with a copyright symbol. 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; see the perlunicode man page for more details.

Note: Prior to AppleScript 1.8.3, do shell script interpreted input and output using the primary system encoding, which made it extremely difficult to deal with files that had non-ASCII characters in their names.

Prior to Mac OS X 10.4, output that is not valid UTF-8 will produce the error "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. As of 10.4, output that is not valid UTF-8 will be interpreted using the primary system encoding.

What are the rules for line endings?

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

I have an AppleScript file or alias object; how do I pass it to a shell command?

The shell specifies files using POSIX path names, which are strings with slashes separating the path components (for example, `/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` class. `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:"
```

POSIX path doesn't work right if the file has certain characters in its name — spaces, parentheses, $, *, etc.

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!"
```

Why doesn't POSIX path just quote everything for me?

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

How do I control an interactive tool like ftp or telnet with do shell script?

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 Perl and most Unix shells.

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.

My script will produce output over a long time. How do I read the results as they come in?

Again, the short answer is that you don't — do shell script will not return until the command is done. In Unix terms, it cannot be used to create a pipe. 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.

I want to start a background server process; how do I make do shell script not wait until the command completes?

Use `do shell script \"command &> file_path &\"`. 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, but see the next question.

Note: Saying `&> file_path` is semantically equivalent to `> file_path 2>&1`, and will direct both standard output and standard error to `file_path`. If you need them to go to different places, direct standard output using `>` and standard error using `2>`. For example, to send standard error to a file but ignore standard output, do this:

```
do shell script "command > /dev/null 2> file_path
```

See the sh man page under "Redirection" for more details.

I have started a background process; how do I get its process ID so I can control it with other shell commands?

You can use a feature of sh to do this: the special variable `$!` is the ID of the most recent background command, so you can echo it as the last command in your shell script, like this:

```
do shell script "my_command &> /dev/null & echo $!"
-- result: 621
set pid to the result
do shell script "renice +20 -p " & pid
-- change my_command's scheduling priority.
do shell script "kill " & pid
-- my_command is terminated.
```

I'm trying to use top, but it fails saying "can't get terminal attributes" or "error opening terminal: unknown."

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.

What's the default working directory for do shell script commands?

do shell script inherits the working directory of its parent process. For most applications, such as Script Editor, this 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.

Does it make a difference which application I tell to do shell script?

For the most predictable results, always put do shell script calls outside of any tell block, or use `tell me`. In practice, it usually doesn't matter unless you are running your AppleScript from osascript, but it

doesn't hurt to be safe. (Also, telling another application to do shell script, or indeed any scripting addition, will block any other work that application might do until the scripting addition finishes. This is often considered a bad thing.)

The issue is that which application you tell determines the environment for the shell script — the working directory, environment variables, and so on. Most applications have the same environment, but relying on this is a maintenance risk. If your AppleScript is running in osascript, the working environment comes from the shell osascript was run from, which is completely separate from any other application. Consider the following exchange in sh running in Terminal:

```
$ VAR=something; export VAR $ osascript -e 'do shell script "echo $VAR"' something $
osascript -e 'tell application "Finder" to do shell script "echo $VAR"' (nothing)
```

For more on how this works, see Gory Details.

Back to Top

Gory Details

This section is for those who want to know all the niggling details of how do shell script works. If you just want to make your script work right, you probably don't need to read it.

What shell does do shell script use, really?

do shell script always calls /bin/sh. However, in Mac OS X, /bin/sh is really a copy of another shell that emulates sh. In 10.2 and later, this is bash; prior to that it was zsh.

How long can my command be, really?

Calling do shell script creates a new sh process, and is therefore subject to the system's normal limits on passing data to new processes: the arguments (in this case, the text of your command plus about 40 bytes of overhead) and any environment variables may not be larger than kern.argmax, which is currently 262,144 bytes. Because do shell script inherits its parent's environment (see the next question), the exact amount of space available for command text depends on the calling environment. In practical terms, this comes out to somewhat more than 261,000 bytes, but unusual environment settings might reduce that substantially.

Where does the shell environment come from — environment variables, working directory, and so on?

do shell script inherits the environment of its parent process. Because of how scripting additions like do shell script work, the parent process is the application for the tell block surrounding the do shell script call. If there is no surrounding tell block, or if you tell me, the parent is the application running the script.

The environment covers the working directory, any environment variables, and several other attributes — see the execve(2) man page for a complete list. As mentioned in Issuing Commands, do shell script does not read the configuration files that an interactive shell running in Terminal would.

Any application launched from the Finder gets the same default environment: a working directory of / and the environment variables HOME, LANG, PATH, USER, and SHELL. (You can define more environment variables if you wish; see Technical Q&A QA1067, Setting environment variables for user processes for details.) Most applications do not change their environment, but relying on this is a

maintenance risk.

osascript(1) inherits its environment from the shell it's run from: the working directory is the working directory of the shell; any environment variables defined in the shell will also be defined in osascript, and therefore in do shell script. For example:

```
$ VAR=something; export VAR $ osascript -e 'do shell script "echo $VAR"' something
```

sh defines a number of environment variables of its own, but those are the same no matter how it's run.

Back to Top

Document Revision History

Date	Notes
2006-03-23	Document security restriction on telling other applications with administrator privileges.
2005-07-20	Updated "administrator privileges" question to reflect changes in 10.4.2.
2005-05-06	Changes for Mac OS X 10.4 (Tiger).
2003-01-27	New document that frequently Asked Questions about the AppleScript "do shell script" command.
