NOTES ON APPLESCRIPT IN TEXSHOP

MICHAEL SHARPE

These are notes made in the course of attempts to write and debug some AppleScripts for TeXShop. The current TeXShop version is 3.25, running under OS X (Mavericks) 10.9. Both Finder and AppleScript (and especially the Finder AppleScript dictionary) can and do change from time to time, and may require rewriting previously functional scripts to accommodate the changes. AppleScript is a head-banger of a language, or perhaps I have to say `language as head-banger`, or `class of AppleScript is head-banger`, or even `language of head-banger as alias`. (In fact, the hyphen in `head-banger` makes it not legal as an AppleScript variable.) The motivating force is that it's the only way to script most Mac GUI applications, like TeXShop.

1. Why Macros?

TeXShop's Macros menu has items whose functions overlap to some extent with engines and their associated executables. (Recall that engines are usually simple shell scripts located in TeXShop's `Engines` folder. These are the possible targets for a line like

```
% !TEX TS-program = latexmk
```

near the top of your tex source file, which instructs TeXShop to process the file with the engine program `latexmk.engine`. The engine files call executable programs located in TeXShop's `bin` folder. (Eg, `latexmk.engine` calls the executable `latexmk`.) If there is an engine available to perform the processing you want, then your experience with the engine will most likely be better than with a macro, for several reasons:

- shell scripts, PERL scripts and, to a lesser extent, python scripts are likely to prove much more robust than AppleScripts, due in part to the changes that occur from time to time in the latter;
- those other script languages are faster than AppleScript and are more intimately connected to the UNIX shell;
- engines and the executables they call will most likely have had much more intensive testing and public exposure, reducing the likelihood of bugs.

Engines like `latexmk` perform all the operations needed, as many times as needed, to produce the final `pdf` output, and there is no longer much need to write AppleScripts to run bibtex and the like. So, in my opinion, the main use for AppleScript macros is operations that either (a) modify your source file but do not involve any TeX processing; (b) provide you with choices of items for your source document; (c) bring up information about command options, etc.

For this reason, I'm going to ignore all verbs in the TeXShop dictionary that initiate document processing (eg, `typeset, latex`) that are best performed by an engine.

Date: November 2, 2013.

2. The ways of using AppleScript macros in TeXShop

Macros are triggered in one of three ways in TeXShop:

- choose the name of the macro from the Macros menu in the document menu bar;
- choose the name of the macro from the Macros menu in the main menu bar on the top line of the screen—in the sequel, this is what we shall mean by the Macros menu;
- press the associated key combination, if one has been defined. (A key combination may be defined using the Macros menu.)

Writing an AppleScript for the Macros menu can be as simple as the following example, which is not at all useful as it stands.

```
--applescript direct
do shell script("/bin/ls") --the parens are optional
```

This is not useful because it does nothing with the output from `ls` and does not check for possible errors. The point is that an AppleScript macro can simply run an external executable program in an arbitrary language. A slightly more realistic example would be

```
--applescript direct
try
    set s to do shell script("/bin/ls")
on error number errnum
    display dialog "Command ls returned error number " & errnum
end try
```

so that the standard output from `ls` is captured in the AppleScript variable `s` and the exit status is passed along to be handled by AppleScript's `on error`. Note that it is important to give a full path to UNIX commands, because you should not make any assumptions about the `path` used by the shell and, in addition, you want to avoid any localizations of the command made by the user so you can be sure the output is in the expected format.

AppleScripts may be involved in the definition of a macro in the Macros menu in four rather different ways.

- **Mode** (1)**:** The AppleScript code is placed in the body of a Macro Editor window following `--applescript direct`. In this case, the code is interpreted by TeXShop's internal interpreter and the TeXShop event loop comes to a halt so that TeXShop does not respond to external stimuli until the event loop restarts. (This could be triggered, for example, by a TeXShop error message popping up.)
- **Mode** (2)**:** The AppleScript code is placed in the body of a Macro Editor window following `--applescript`. In this case, the code is interpreted by a copy of ScriptRunner in the TeXShop bundle. The TeXShop event loop continues but messages from ScriptRunner need to be brought to the front with an `activate` command or may be easily missed.
- **Mode** (3)**:** This refers to AppleScript code that is in an external compiled AppleScript which is loaded by your macro script as an external AppleScript library. In this case, handlers in the external library are not subject to the limitations of `--applescript [direct]`. This may be the optimal strategy, at the cost of having to write replacement macros for methods such as `save`, `close` and `open` which can be problematic in mode (1). Note though that a handler in an external library that has commands from the TeXShop dictionary in a `tell "TeXShop"` block will behave

the same way as the same commands in a similar block in the main script. The main advantage of external scripts is that the error handling can be written just once.

- **Mode** (4)**:** Save an AppleScript as an application called by `do shell script`. The interface does not differ in any way from applications written in other languages, and we shall not discuss it any further except to say that other languages are usually much more capable and better-suited for stand-alone use.

3. General advice on writing AppleScript macros for TeXShop

Please understand that the following advice applies only to use in TeXShop—in other settings, it might not be helpful.

- Do all work with files using names in POSIX string format. This will keep the number of conversions and coercions to a minimum. You need to be aware of the difference between a simple POSIX string (eg, `set fstr to "/Users/Al/Documents"`) and the result of

  ```
  set f to (POSIX file fstr)
  ```

  which is no longer a string, but an object of type `file`. The latter can be coerced to the former by

  ```
  set fstr to (POSIX path of f)
  ```

- One of the first things to do, early in your script, is to gives names to some key folders to permit easier path construction. Eg,

  ```
  set docf to POSIX path of (path to documents folder)
  set homef to POSIX path of (path to home folder)
  ```

  (You should never use ~ as an abbreviation for your home folder as it can fail.)

- Make use of the special constants TeXShop provides. Most useful are `#DOCUMENTNAME#` and `#FILEPATH#`. See the full list and description in Section .

- To open a file named `eg1.txt` in your Documents folder for writing output:

  ```
  set txtfile to docf & "eg1.txt" --docf ends with /
  set outf to (open for access (POSIX file txtfile) with write permission)
  write "xyz" to outf
  close access outf
  ```

  To read from that same file:

  ```
  set inf to (open for access (POSIX file txtfile))
  set thetext to (read inf for (get eof inf)) --Mac OS Roman
  close access inf
  ```

- Make use of `TSLib.scpt`, which contains a number of workarounds for TeXShop scripting issues. (It is described in detail later in these notes.)

- Mode (1) is a little easier to work with than Mode (2), but suffers from some serious limitations when you want to open, close or save files without making use of TSLib. Use Mode (2) if you need to perform extensive file manipulation while keeping TeXShop responsive to changes.

- The command `do shell script` offers many possibilities but there is an overhead cost when using it for small external scripts. Whenever possible, pipe commands together within one `do shell script` to avoid repeated opening and closing of shells.
- AppleScript is fine for small jobs, but large amounts of data should be handled externally.
- Whenever possible, make changes to a source file by working with the `selection`, adjusting its `offset` and `length` and then manipulating its `content`. The `search` handler can be of much use in this connection.

4. Starting a new script

Unless your script is only a few lines long, you will find it much easier to develop it using the AppleScript Editor to get the basic layout correct. It provides specific (though sometimes puzzling) error messages, but provides the exact location of the error, unlike the TeXShop Macro Editor. The fly in the ointment is that from the AppleScript Editor you cannot make use of some of the best features of TeXShop's AppleScript interpreters—the special constants with names like `#FILEPATH#`, which greatly simplify the assembly of filenames and window names. I find it useful to work as follows.

- Decide which constants you will need. At a minimum, you will most likely make use of `#FILEPATH#`, `#NAMEPATH#` and `#DOCUMENTNAME#`, so in the Macro Editor, make your first lines following the initial comments

  ```
  set filepath to #FILEPATH#
  set namepath to #NAMEPATH#
  set docname to #DOCUMENTNAME#
  ```

  and make no further use of the `#..#` constants in your script. Then, in the AppleScript Editor, place the corresponding definitions at the top of the file.

  ```
  set filepath to "/path/to/tex_file"
  set namepath to "/path/to/tex_file_less_extension"
  set docname to "name_of_frontmost_tex_document"
  ```

  so they correspond exactly to the output from `#FILEPATH#`, `#NAMEPATH#` and `#DOCUMENTNAME#`.
- The remainder of your script (call it the body) should be developed in AppleScript Editor to take advantage of its better error diagnoses. Copy the body of the finished script from AppleScript Editor when it works there, then focus on issues that arise in Macro Editor. (See below.)

Because of the problems with AppleScript macros running in the Macro Editor (documented below), I find it useful to make use of an external library of handlers which, being compiled outside TeXShop, do not have the same problems, acting as if they were running in AppleScript Editor. (An exception is commands running inside a `tell "TeXShop"` block, which have the same issues they would in the main script.) To load the routines, place these lines following the lines discussed above.

```
set TSLibAlias to alias ((path to home folder as string) &
  "Library:TeXShop:Scripts:TSLib.scpt") --join to previous line
set TSLib to (load script TSLibAlias)
set mytex to POSIX path of (path to documents folder) & "texfiles/"
```

The last line is not needed to load the library, but makes it easy for me to construct the POSIX paths to files in that folder.

Here then are how my templates appear:
In Macro Editor:

```
--applescript direct
--possibly without "direct"
set filepath to #FILEPATH#
set namepath to #NAMEPATH#
set docname to #DOCUMENTNAME#
# CUT HERE
# Start of script body
set TSLibAlias to alias ((path to home folder as string) &
   "Library:TeXShop:Scripts:TSLib.scpt") --join to previous line
set TSLib to (load script TSLibAlias) --to use handlers in TSLib
set mytex to POSIX path of (path to documents folder) & "texfiles/"
```

In AppleScript Editor:

```
set filepath to "/path/to/tex_file"
set namepath to "/path/to/tex_file_minus_extension"
set docname to "name_of_document"
# CUT HERE
# Start of script body
set TSLibAlias to alias ((path to home folder as string) &
   "Library:TeXShop:Scripts:TSLib.scpt") --join to previous line
set TSLib to (load script TSLibAlias) --to use handlers in TSLib
set mytex to POSIX path of (path to documents folder) & "texfiles/"
```

`TSLib.scpt` contains the following handlers. In all cases, the arguments should be strings. Arguments representing files should be in POSIX path form and the names of documents in TeXShop should be as (used to be) displayed in the TeXShop document window, just as provided by `#DOCUMENTNAME#`. Note that it not safe to use the abbreviation ~ for the home folder in specifying a POSIX path. Use instead

```
POSIX path of (path to home folder)
```

for the equivalent POSIX path, a string ending with '/'.

All handlers involving files or folders return the the number 1 on failure. Those that do not return a string value return the number 0 on success.

- `opendoc(f)` opens the POSIX path `f` in TeXShop. It replaces the troubled `open()` handler in TeXShop. Eg, `opendoc(mytex & "eg.tex")`.
- `savedoc(f)` saves document `f` in TeXShop using its current location. An error is returned if it has not been saved previously. Eg, `savedoc("eg.tex")`.
- `savedocIn(f,g)` saves document `f` in TeXShop to a POSIX file `g`, overwriting `g` if it exists. Note that the document `f` is not closed and file `g` is not opened. Eg, `savedocIn("eg.tex",mytex & "eg1.tex")`.
- `closedoc(f)` closes the TeXShop document named `f`. If the document has been modified since it was last saved, it will be saved under its own name. Eg, `closedoc("eg.tex")`. If the document has never been saved (eg, "Untitled") and has been modified, you will be asked for a name and can choose not to save it.

- `closedocIn(f,g)` closes the TeXShop document named `f`, saving its contents in the POSIX file `g` but not saving changes to the original file. Eg,

  ```
  closedocIn("eg.tex",mytex & "eg1.tex")
  ```

- `docexists(f)` takes the POSIX path `f` and returns the number 0 if it specifies an existing file or folder, 1 if not.

- `dirbase(f)` takes the POSIX path `f` and returns a list with two items, the first being the POSIX path to the parent folder, the second the name of the file within the parent folder. These should give the same output as the UNIX commands `dirname` and `basename` without the overhead of `do shell script`.

- `docname(f)` takes the POSIX path `f` of an existing `.tex` file and returns a list with three items, the first being the POSIX path to the parent folder, the second the name of the file within the parent folder with `.tex` removed, and the third provides the name by which the document *should* be known if opened in TeXShop. For example, assuming `"/Users/Al/Documents/eg.tex"` exists, `docname("/Users/Al/Documents/eg.tex")` would return

  ```
  {"/Users/Al/Documents","eg","eg.tex"}
  ```

  if the file had not been saved with extension hidden, and otherwise it would return

  ```
  {"/Users/Al/Documents","eg","eg"}
  ```

  This gives you an easy way to construct the equivalents of `#..#` items when you open a new `.tex` file in a script. Eg, having specified an existing tex file with POSIX path string s,

  ```
  tell TSLib to set {pdir, shortname, displayname} to docname(s)
  set dviname to pdir & "/" & shortname & ".dvi"
  ```

- `mkdir(f)` traverses the POSIX path `f`, recursively creating any folders necessary. Every component of `f` will be created as a folder, if such a folder is not present, so do not pass the POSIX path to a file. Eg

  ```
  mkdir("/x/y/z")
  ```

  will try first to create the folder `x` in the root folder. (This will fail because users do not have permission to write to the root folder.) However, had it succeeded, it would then create a subfolder `y` of `/x` and a further subfolder `z` of `/x/y`. It would not overwrite any existing folder with a new, empty folder, so it is a safe command.

- `stroffset(a,b)` is a synonym for the usual `offset of a in b` which is unavailable in modes (1,2) because the meaning of `offset` has been pre-empted there. Eg, `stroffset("b","abc")` returns 2.

- `trim(s)`, where `s` is a string, removes white-space characters (space, tab, line-feed) from both ends of `s`. Actually, you can specify what to remove by temporarily changing the value of `trimitems`. From modes (1,2), you could use

  ```
  set TSLib's trimitems to {ASCII character 10, ASCII character 0,tab}
  tell TSLib to set s to trim(s)
  set TSLib's trimitems to origtrimitems  --reset
  ```

5. TeXShop constants

In modes (1,2), one has access to special named constants described below. **CAUTION:** these constants do not function while the Macro Editor is open, making it impossible to use the Test button if the script depends on those constants. The values depend on which window is frontmost in TeXShop at the instant the script started. Let's say that the front window is any one of `eg.tex` (or just `eg`, if you saved the file with hide extension checked) or `eg.pdf` or `eg console`. Then the following constants are defined:

- `#FILEPATH#` is the full POSIX path string to the file `eg.tex`;
- `#TEXPATH#` is the same as `#FILEPATH#`;
- `#PDFPATH#` is the full POSIX path string to the file `eg.pdf`, if it exists;
- `#DVIPATH#` is the full POSIX path string to the file `eg.dvi`, if it exists;
- `#PSPATH#` is the full POSIX path string to the file `eg.ps`, if it exists;
- `#LOGPATH#` is the full POSIX path string to the file `eg.log`, if it exists;
- `#AUXPATH#` is the full POSIX path string to the file `eg.aux`, if it exists;
- `#INDPATH#` is the full POSIX path string to the file `eg.ind`, if it exists;
- `#BBLPATH#` is the full POSIX path string to the file `eg.bbl`, if it exists;
- `#HTMLPATH#` is the full POSIX path string to the file `eg.html`, if it exists;
- `#NAMEPATH#` is the full POSIX path string to the file `eg.tex`, minus the `.tex`;
- `#DOCUMENTNAME#` is the name as it *should* appear in the source window, either `eg` or `eg.tex`, depending on whether the document was saved with hide extension checked or not. (This used to be true, but is not the case at the moment. Nonetheless, it is the name used internally to refer to the document.)

To emphasize what is perhaps an obvious point, these are fixed for the duration of the script and may not reflect current values correctly if windows were closed or opened by the script.

6. Working with a Selection

This is not quite as obvious as it seems. A text selection has an offset, a length and a content, and you should think of these as a snapshot of the current selection at the instant you read a selection, but which will change whenever you modify (ie, `set`) any one of them. Note that offset 0 corresponds to a cursor position immediately before the first character of the document and, if the document has N characters (including EndOfLine characters), an offset value of N corresponds to the cursor immediately after the last character of the document.

When you set a selection property, the other properties may also change. Eg, with

```
tell application "TeXShop"
    set offset of selection of document docname to n --0\le n\le N
end tell
```

the effect is:

- the beginning of the selection changes to offset `n`—if `n` is set to a value out of range, the length is set to 0 and the offset to N, so a good way to send the cursor to the end of the file is:

```
tell application "TeXShop"
    set offset of selection of document docname to -1
end tell
```

If you then ask for the offset of the current selection, you will get the number of characters in the document, which is otherwise not so obvious to determine. You might think

```
tell application "TeXShop"
    set n to count of characters of document docname
end tell
```

might work, but the TEXShop dictionary does not know about `characters of document ...`. You can write

```
tell application "TeXShop"
    set n to count of characters of text of document docname
end tell
```

but that is highly inefficient for long documents.

- the length stays the same except for an adjustment so the end of the selection stays in range—ie, the `length` will change to `newlength=N-n` if `N-n < length`
- the `content` changes to the text fragment of the document from `n` thru `n+newlength`.

Similarly, with

```
tell application "TeXShop"
%    set length of selection of document docname to k --k\ge 0
%
end tell
```

the effect is:

- the beginning of the selection is unchanged if `k >= 0`, but effectively, nothing is selected (cursor not visible) if `k < 0`;
- the end changes to `newend=Min(offset+k,N)`;
- the `content` changes to the text fragment of the document from `offset` thru `newend`.

Likewise, with

```
tell application "TeXShop"
    set content of selection of document docname to s
end tell
```

the effect is:

- the offset of the selection is unchanged;
- the end changes to `offset+(count of s)`;
- the `content` changes to s;
- that is, the previous selection is replaced by s, and its length is modified accordingly.

The interaction with TeXShop's `goto` command calls for some clarification. First of all, it fails in Mode(1). Assuming now that we are not using Mode(1), if there are `k-1` linefeeds in a document, there are `k` lines with indices `1..k`. The effect of

`tell application "TeXShop" to tell document docname to goto line j`

depends on the location of the current selection.

- If `j<1` or `j>k-1`, there is no effect. (Note: `goto` cannot be used to move to the last line of a document. You may use `set offset to -1`, as described above.)
- If the current selection is completely contained in line `j`, the selection remains unchanged;
- In all other cases, provided `0<j<k` the selection changes to all of line `j` including the EndOfLine character.

Suppose you want to expand the current selection to include every complete line touched by the selection. The following would handle the job.

```
set docname to #DOCUMENTNAME#
set lf to (ASCII character 10)
tell application "TeXShop"
    set offs to offset of the selection of document docname
    set thelast to (length of the selection of document docname) + offs
    set strt to 0
    if offs > 0 then
        set strt to search document docname for lf with searching backwards
    end if
    set offset of selection of document docname to -1
    set endofdoc to offset of selection of document docname
    set offset of selection of document docname to thelast
    set length of selection of document docname to 0
    set theend to endofdoc
    if thelast < endofdoc then set theend to search document docname for lf
    if theend = 0 then set theend to endofdoc
    set offset of selection of document docname to strt
    set length of selection of document docname to theend - strt
    set s to content of selection of document docname
end tell
```

Note that `search` is 1-based, not 0-based and to get the result you expect, you may need to subtract 1 in some cases to get the `offset` right. (Not so in the example above.)

7. Problems with document names

The document name is important in addressing a TeXShop window correctly. If you are in a position to use `#DOCUMENTNAME#`, that will serve for addressing the tex source window. Currently, it may not in fact be the title of that window in TeXShop. If you saved a tex `eg.tex` file with hidden extension, then in earlier version of Mac OS X and TeXShop (several years ago), the following took place:

- the file system would use eg.tex as the name of the file;
- the name would display as eg in a Finder folder window;
- the AppleScript Finder command `exists eg` would return `true` and `exists eg.tex` would return `false`;
- TeXShop would open the file with title **eg**, not **eg.tex**, and would refer to the document by the name `eg`.

The current behavior is different:

- the file system would continue to use eg.tex as the name of the file;
- the name would still display as eg in a Finder folder window;
- the AppleScript Finder command `exists eg` would return `false` and `exists eg.tex` would return `true`;
- TeXShop would open the file with title eg.tex, not eg, but would continue to refer to the document by the name eg, not `eg.tex`.

This change breaks the external library `setname.scpt` by Claus Gerhardt that used to provide the document name but which no longer a gives correct result when a tex file was saved with a hidden extension.

Fortunately, the `#DISPLAYNAME#` constant makes it unnecessary to use this external library in case you work with files that are already open and are frontmost in TeXShop, and the library `TSLib.scpt` contains a suitable replacement, `docname`, that can be used if your script opens a new tex file or brings a different window to the front. The replacement handler consults the file metadata for the correct display name. You may also use a construction like

```
tell application "Finder" to set dn to displayed name of file POSIX file f
```

(Oddly, the `file` preceding `POSIX file` really is necessary.)

8. Specific Problem Areas

8.1. Problems with `--applescript` mode.

There are three main problems aside from the ones mentioned in the preceding subsection.

- The usual AppleScript line continuation character generated by Option-lower case L is not recognized as such, though it is in `--applescript direct` mode.
- The method by which messages are passed back to the user are not reliable. Despite the presence of an `activate` line near the top of the script, errors that occur may not come to the front and there may be no visual indicator that an error occurred unless you think to check the ScriptRunner icon in the dock.
- You may realize after a while that there are several copies of ScriptRunner working at one, each with its own error message, and it is at least easier to shut them all down than in mode (1).

Principally for the second reason, I try to use only the `--applescript direct` mode even though its understanding of the TeXShop dictionary is more limited, because it is not hard to work around those limitations using the external library `TSLib`.

8.2. Problems with `--applescript direct` mode.

It's possible to get into a truly puzzling mess if the script crashes before completion and there are files left open at the time of the crash. (I'm talking about files you've opened using something like

```
open for access mylog with write permission
```

and are written to periodically, like a log file.) What happens in this case is repeated error message (when trying the execute the line above) that the file is already open. The only solution is to close the TeXShop application and restart it. This may be the downside to having TeXShop running the AppleScript interpreter.

It would seem to be good practice to save log messages in a list, then write the list to file at the end so there are no interruptions and the file can be closed as quickly as possible.

In this mode, many forms of save, open and close do not work, or work but produce an AppleEvent time out error. For this reason, you should always either (a) filter out error −1712 in a try .. on error block, or use the corresponding handler from TSLib. For specific problems, see the next section..

8.3. **Problematic commands in TeXShop's AppleScript dictionary.**

Here is an example of the defensive code you should employ in mode (1) where save, open and close can produce spurious AppleEvent timed out errors (error number −1712.)

```
-- assumes f in POSIX string format
set cdate to (current date) + 30 -- allow 30 seconds for timeout
tell application "TeXShop"
    try --the following form works in all modes
        open f as POSIX file
    on error errmsg number errnum
        if (errnum = -1712) and ((current date) < cdate) then
        --applescript direct often provokes error number -1712
            --set errmsg to ""
        else
            display dialog errmsg
        end if
    end try
end tell
```

In all examples and tests below, it is assumed that there is an existing tex file f specified by

```
set f to (POSIX path of (path to documents folder) & "eg.tex")
```

and a (possibly non-existent) file fn defined by

```
set fn to (POSIX path of (path to documents folder) & "eg1.tex")
set pfn to POSIX file fn
```

Let dn be the name of a document open in TeXShop.

The column heading Mode1N stands for Mode (1) with no errors, Mode1E stands for works in Mode (1) but provokes AppleEvent timed out error, while OtherModes stands for works in Modes (2,3).

Save

Command	Mode1N	Mode1E	OtherModes	Remarks
save dn	✓		✓	Not saved unless modified
save document dn		✓	✓	Saves regardless
save front document		✓	✓	Saves regardless
tell document f to save		✓	✓	Saves regardless
save document dn saving in pfn		✓	✓	
save dn in pfn				Did not work, no error msg
tell TSLib to savedoc(dn)	✓		✓	
tell TSLib to savedocIn(dn,fn)	✓		✓	

Open

Command	Mode1N	Mode1E	OtherModes	Remarks
`open pfn`		✓	✓	
`open fn as POSIX file`		✓	✓	
`tell TSLib to opendoc(fn)`	✓		✓	

Another option is to use the UNIX `open` command (works in all modes)

```
open -a "TeXShop" '/Users/Joe/Documents/test.tex' --(via do shell script)
```

but this has some problems in mode (1) because the TeXShop event processing loop is suspended—TeXShop does not see that the file has been opened, and does not modify its document list until TeXShop once again receives the focus. This can lead to very puzzling behavior in scripts.

Close

Command	Mode1N	Mode1E	OtherModes	Remarks
`close dn`				Does not close, no error msg
`close document dn`		✓	✓	Saves if modified
`close front document`		✓	✓	Saves if modified
`tell document f to close`		✓	✓	Saves if modified
`close document dn saving in pfn`		✓	✓	Saves to fn, not dn
`save dn in pfn`				Did not work, no error msg
`tell TSLib to closedoc(dn)`	✓		✓	Saves if modified
`tell TSLib to savedocIn(dn,fn)`	✓		✓	Saves to fn, not to dn

Other supposedly possible forms, like

```
close document dn saving ask
```

provoked an AppleEvent timed out error in mode (1), but saved the changes, and did not ask at all in either of modes (2,3), so its effect is identical there to `close document dn`.

Count

This works as you would expect in all modes and makes a satisfactory replacement for the non-functional `length` command.

```
set n to count documents
```

results in a count of all tex source documents currently open in TeXShop.

Document

This behaves mostly as you would expect, with one peculiarity in mode (1).

```
set doclst to documents --returns list of all open tex source documents
```

Within the same TeXShop tell block,

```
repeat with f in doclst
    set s to (name of f)
    set p to (path of f)
    set b to (modified of f)
end repeat
```

all function correctly, but outside the TeXShop tell block, the first fails in mode (1).

Search

The `search` method is not problematic—it seems to work correctly in all modes but the documentation is a bit sparse and the meaning is slightly unintuitive. It is called with a line like

```
search document "eg.tex" for "\\begin{" --need to escape backslashes
-- additional options as below
-- [case sensitive <boolean>] : if omitted, default value false
-- [matching as whole word <boolean>] : if omitted, default value false
-- [searching backwards <boolean>] : if omitted, default value false
-- [starting from <integer>] : if omitted, beginning of current selection.
```

The returned value is an integer, the index (starting from 1) of the first character of the found string—0 if not found. (If found, this is the offset of the found string +1.) This method modifies neither the current selection nor the cursor position. Because this search method uses native TeXShop code rather than AppleScript string search, it should be more efficient in practice, and with a couple of searches one may build a `selection`, from which the `content` may be extracted. One more thing to keep in mind with a search:

- with a forward search, the search begins at the cursor (ie, the offset of the selection), but with a backward search, the search starts at the character before the cursor. Eg, if the document looks like `12|34` (with `|` representing the cursor), then

  ```
  tell application "TeXShop"
      search document docname for "3"
      search document docname for "3" searching backwards
      search document docname for "2" searching backwards
  end tell
  ```

  results respectively in 3, 0, 2.

The unintuitive part is what happens at the beginnings of lines. Suppose the document has just five characters, laid out like

```
12
|45
```

where `|` represents the cursor and the character with index `3` is the linefeed character with ASCII number 10. The `offset` would report that the cursor is at position 3, and if you search forwards for the next linefeed, the result is `3` again. To find the end of the line, you need to start at a position 1 past the cursor in this case.

9. Using refreshtext

If you change the contents of a TeXShop file within a macro headed

```
--AppleScript direct
```

you can't use `refreshtext` with any effect because TeXShop's event loop is suspended. This leave the document without syntax coloring, for example. One solution is to run as the last command of a script an external command that instructs TeXShop to run `refreshtext` on the front document. To do this, I make a shell script named `refreshfront` that I saved in `~/bin` with contents

```
#!/bin/bash
exec osascript <<END
```

```
tell application "TeXShop"
  tell front document to refreshtext
end tell
END
```

This script has to be made executable with the command

```
chmod 755 ~/bin/refreshfront
```

To call this from your macro, the last executed line of the macro should be

```
do shell script "~/bin/refreshfront &> /dev/null &"
```

which seems to return control to TeXShop without waiting for the shell script to finish execution, so that TeXShop's event loop is running when the osascript completes.

10. Appendix: File paths, AppleScript aliases and Path references

AppleScript macro writers need to have a firm grip on the differences between file paths, AppleScript aliases and AppleScript *Path references*, the first form being strings in either traditional HFS format like `"Macintosh HD:Users:"` or POSIX format like `"/Users/"`. AppleScript aliases are a bit more slippery. To create an AppleScript alias:

```
alias "Macintosh HD:Users:" --provided this folder exists
```

AppleScript will raise a run-time error if the file path you specify following `alias` does not resolve to an existing file or folder. So, an AppleScript alias is really a form of reference (ie, a pointer) to an existing file or folder—one that is understood by all scriptable applications. The need for a method of referring to an incipient file or folder is clear, but the means of doing so is somewhat less so. AppleScript uses (or used to use) the term *Path reference form* for an object whose text representation is like

```
file "Macintosh HD:Userz:"  --the folder need not exist
```

but the file object cannot be created just by entering that expression in an AppleScript. The peculiar thing about these objects is that they seem to be permitted to be constructed only in special situations that call for them. The output from a Choose File Name dialog is just such an object. For other examples, in the AppleScript Editor, the following all work:

```
set newf to POSIX path of (path to documents folder)&"newfile.txt" as file specification
POSIX path of file "Macintosh HD:Userz:"
  -- result is incorrect "/Macintosh HD/Userz/" if no "Macintosh HD:"
  -- otherwise result is "/Userz/"
tell application "TeXShop"
    save front document in file ((path to documents folder as string) & "eg.tex")
end tell
tell application "TeXShop"
    save front document in (file ((path to documents folder as string) & "eg.tex"))
end tell
tell application "Finder"
    if the file theFile exists then set x to 1 -- theFile in HFS format
end tell
```

even though `file ((path to documents folder as string) & "eg.tex")` produces an error if run by itself. What seems to be happening in some cases is that a Path reference may be accepted in place of an AppleScript alias even when the dictionary documentation specifies the need for an alias.

For the TeXShop scripter, the most important means of creating a Path reference is `POSIX file`, which always works. For example

```
POSIX file "/Users/" --result like 'file "Macintosh HD:Users:"'
POSIX file "/Users" --result like 'file "Macintosh HD:Users"'
```

which may be coerced to AppleScript aliases, if they exist, by appending `as alias`.

It may be helpful to think of an alias as something you can copy *from*, and a Path reference as something you can copy *to*.

By and large, POSIX style filenames are useful only when working on the UNIX side, where applications understand neither aliases nor the HFS style file path, or when using handlers in `TSLib` which expect files to be specified in that form.

To convert between these formats is usually simple but slightly odd in some cases:

- `alias "Macintosh HD:Users:" --create an alias from a folder name`
- ```
  POSIX path of "Macintosh HD:Users:"
     -- result is incorrect "/Macintosh HD/Users/" if no "Macintosh HD:"
     -- otherwise result is "/Users/"
  ```
- ```
  POSIX path of file "Macintosh HD:Users:"
     -- result is incorrect "/Macintosh HD/Users/" if no "Macintosh HD:"
     -- otherwise result is "/Users/"
  ```
- ```
  POSIX path of alias "Macintosh HD:Users:" --returns "/Users/" if it exists
     --error if non-existent
  ```
- `"/Users/" as POSIX file --result like 'file "Macintosh HD:Users:"'`
- `"/Users/" as POSIX file as string --result like "Macintosh HD:Users:"`
- `"/Users/" as POSIX file as alias --result is 'alias "Macintosh HD:Users:"'`
- `<any alias> as string --result is like "Macintosh HD:Users:"`
- `POSIX path of (file "Macintosh HD:Users:" as string) --fails`
- `POSIX path of (file "Macintosh HD:Users:") --works`
- `file "Macintosh HD:Users:" --fails`

When passing a POSIX style path to a UNIX command, one should guard again the possibility of spaces somewhere in the POSIX path by referring to the `quoted form of`:

```
set ppath to POSIX path of "Macintosh HD:Users:Joe Blow:"
quoted form of ppath  --result is '/Users/Joe Blow/'
```

10.1. **Checking for existence.** Finder has an `exists` method which must be called as part of a Finder tell block. The method has one problem—the name used by Finder may be different from the name shown in a Finder window if the file was saved with hidden extension.

10.2. **Creating a chain of folders.** AppleScript's syntax is painful if you need to create a deeply nested chain of folders. The `mkdir -p` command from `bash` (AKA `sh`) does the work efficiently, creating the entire chain of nested subfolders, as necessary.
```

```
--expects a folder f specified in POSIX form
try
    do shell script("/bin/mkdir -p " & quoted form of f)
on error errmsg
    display dialog errmsg
end try
```

It's important to catch a possible error with an `on error` fragment, as it may be the only way to know whether the command succeeded.

10.3. **Finder operations.** If you need to deal with files and folders using Finder, you may need to be aware of some terms from its dictionary. In Finder, you use terms

```
folder "Macintosh HD:Users:joe:" --note final :
```

that are special to Finder and must be wrapped in a Finder tell block. There are also useful terms for special locations in your file system, all returning Finder aliases:

```
home, home folder -- returns something like 'folder "Macintosh HD:Users:joe:"'
desktop
startup disk  -- returns something like 'folder "Macintosh HD:"'
```

all of which may be coerced to string form by appending `as string` or used directly in Finder operations.

Finder example:

```
tell application "Finder"
  if not (exists folder "TeXShop_test" of home) then
    make new folder at home with properties {name:"TeXShop_test"}
  end if
  set the_folder to POSIX path of ((folder "TeXShop_test" of home) as string)
end tell
```

`StandardAdditions.osax`, which is loaded automatically, also defines `path to` with defined locations including

```
path to application support
path to applications folder
path to documents folder
path to downloads folder
path to favorites folder
path to Folder Action scripts
path to fonts
path to home folder
path to library folder -- like 'alias "Macintosh HD:Library:"'
path to library folder from user domain
  -- returns alias to your home library provided it is visible in Finder
path to preferences
path to public folder
path to shared libraries
path to system folder
path to system preferences
path to temporary items
```

```
path to users folder
```

The result in each case is an AppleScript alias, which may coerced to a string by appending `as string`. The commands need not be run only in a Finder tell block.