About Scripts

Introduction

A script is a sequence of commands that SerialEM can execute. A script can perform all of the individual actions involved in acquiring a tilt series, such as tilting, autofocusing, autoaligning, and saving images. In addition, there are many commands for other actions such as moving the stage or changing magnification or defocus.

Scripts were called 'macros' until SerialEM 3.6. You may still see the term 'macro' used in various places, or in older scripts.

Scripts can automatically repeat themselves and they can contain internal loops that are executed a specified number of times. One script can call another like a subroutine; when the second script finishes, the next command in the calling script will be run. Functions can also be defined and called, but in a more flexible fashion than calling a script. There can also be IF statements for conditionally executing some statements. Loops and IFs together may be nested to 40 levels deep and calls may be nested to 40 levels deep. A script can also contain a statement at its end to switch to running another script, but this is not allowed in a script called by another script.

The commands are typed into a Script Editing Window, which can be opened either from the Script menu or by holding down the Ctrl key while pressing an enabled script button in the Camera & Script Control Panel or in the toolbar that can be opened from the Script menu. One command is entered per line. Blank lines and lines starting with # are ignored. Comments can also be placed after the command on a line, starting with # after a space.

Scripts can be run from the Script menu, from the Camera & Script Control Panel, or from the Script toolbar. In addition, CTRL F1 through CTRL F10 are hotkeys for running the first 10 scripts. A script will stop running if various conditions specified in the Script Controls dialog box are not met, such as the minimum number of counts in an image.

When a script is started, the program will first check through the script, and through any other scripts that it calls. It will check that the script(s) contain defined commands, that loops and IF blocks are properly nested and terminated, and that various commands have the required entries after them. If it finds an error at this phase, it will report the error along with a statement that nothing has been executed. Only some types of errors are found at this stage; other errors will not be found until the bad statement is reached during execution.

A script can define a name that will be displayed in several places: on the title bar of the editing window, in the Camera and Script Control Panel when one of the script running buttons is dialed to that script, in the script toolbar, and in the Run submenu of the Script menu. The name should be short to fit into the script buttons. It can contain spaces. A longer name can be defined to display in the Run submenu. When a name is first added to a script, it will appear after some action is taken with the script.

Scripts will be saved when Settings are saved, and when a settings file is loaded, any scripts in that file will replace scripts already in the program.

Using Variables

Variables can be defined in scripts and some arithmetic can be done in a line that defines a variable. There are six kinds of variables: regular ones defined on a line with an equals sign; persistent ones defined on a line with ':=' which will persist from one script run to another; loop index variables defined in a LOOP statement; variables set when values are reported with almost all of the 'Information Output Commands' listed below; and regular or persistent string variables that are defined on a line with '@=' or ':@=', respectively. A variable can be used in any command by putting a '$' in front of the variable name; all commands are scanned first to substitute the values of variables. The variable can be embedded in a text string, such as for opening a file, without needing to be separated by spaces from the rest of the string. Note that like commands, variables are not case-sensitive and can be referred to as '$Name', '$name', '$NAME', etc.

1. A regular variable can be defined with an expression of the form 'Name = value'. In general, everything in such an expression must be separated by spaces. There must be spaces between the name and the equals, and between the equals and the value. The value can consist of an arithmetic expression, which is processed and reduced to a single value (see below). Only the first value or word of text after the equals sign is stored as the value; additional text is ignored. An existing variable can be given a new value, but it must be assigned to in the same way; i.e., with '=' for a regular variable or ':=' for a persistent variable. If the value is text rather than a number, such as a filename, its case is preserved only as of SerialEM 4.6.0 (Sept. 10, 2016); before that, the text was converted to upper case. If you rely on case being preserved in variable assignments and there is some chance of your script being shared and tried on earlier versions of SerialEM, you should include a line
   ```
   Require keepcase
   ```
   somewhere in your script.
2. A persistent variable can be defined with an expression of the form 'Name := value'. The ClearPersistentVars' command can be used to remove all persistent variables, or an individual variable can be cleared by leaving out the value, as in 'Name :='.
3. A regular or persistent string variable can be defined with a expression of the form 'Name @= string including everything else on the line' or 'Name :@= string', respectively. Variables will be substituted but the text on the right will not be evaluated in any other way, and the entire text after the '@=' will be assigned to the variable. This would be useful for preserving spaces in a filename that has been defined by the user.
4. A loop index can be defined by adding an unused variable name after the number on a LOOP line. This variable will have a value from 1 through the loop count. It is maintained by the program and cannot be assigned to. The variable becomes undefined after the loop ends and can then be reused as a loop index or defined variable.
5. Other than loop indexes, all variables are 'global' by default, which means that they can be

accessed or reassigned from any script or function, regardless of where they are first created. Local variables that are accessible only within the script or function where they are defined can be created with the command

```
LocalVar varName1 varName2 varName3 ...
```

The listed variables will not be seen as defined and cannot be changed in any other functions or scripts, including functions in the same script. They will become undefined when returning from function or script. They must not already exist as global variables created in the current function or script. If the variable name has been defined elsewhere, the local copy will be kept separate and not affect the global value. Loop indexes can also be made local with the 'LocalLoopIndexes' command.

6. When a 'Report...' command is used, the values reported are generally assigned to variables named 'ReportedValue1', 'ReportedValue2', and so on up to 6, as well as to shorter variables named 'RepVal1', etc. This assignment can be assumed when the command document is silent about it. Some other commands set these variables as well; in these cases the command documentation should specify what is set.
7. For almost all 'Report...' commands, and a few others that set 'ReportedValue' variables, one or more variable names can be placed at the end of the command, and the reported values will also be assigned directly to those variables. This particularly useful for saving values that need to be restored later. There need not be a variable provided for every value being reported. For example

```
CameraProperties sizeX sizeY
```

will assign the X and Y size of the current camera to 'sizeX' and 'sizeY' as well as to the regular 'ReportedValue' variable, and assigns a third output, the RotationAndFlip property, only to 'RepVal3' and 'ReportedValue3'. Note these points:
 * 'Report...' commands will assign to listed variables unless the command documentation says 'No optional assignment to variable.' These will generally be commands with optional entries.
 * Some commands that take optional entries will assign to listed variables, but the optional entry is required in order for this to work. These will generally be commands that take a buffer letter, where it is easy for the program to tell if you have omitted the letter and listed a variable instead. The command documentation will say 'Values can be assigned to variables after a buffer letter.'
 * Commands other than 'Report...' commands will not assign to listed variables unless the command documentation says so, such as with 'Values can be assigned to variables at end of command.'
 * If you use this feature and there is any possibility that the script will be distributed and possibly run on earlier versions of SerialEM, you should include the command

```
Require variables1
```

somewhere in your script. Earlier versions will not pay any attention to variables after commands and will just not work correctly, so this command is a good way to catch the problem. (This feature was added to the 3.6.0 beta version on about March 10, 2016).

Array Variables

Variables can also be set to an array of values, and individual values can be accessed by their index in the array. To assign an array to a regular or persistent variable, enclose the value in curly braces, '{' and '}', for example:

```
    exposures = {0.5 1.0 1.5 2.0}
```

Space between the '{' or '}' and the values is optional. Variables can be included in the list of values. Some arithmetic expressions can also be included, provided that each element requiring evaluation is enclosed in parentheses. For example:

```
    exposures = {$base ($base + 0.5) ($base + 1.0)}
```

These values can then be accessed with '$exposures[index]', where the index is numbered from 1. The index can be a variable; it can even be an indexed array variable; but no arithmetic can be done inside the brackets. The index must be close to an integer and between 1 and the number of elements in the array. Thus,

```
   $exposures[2]
   $exposures[$loopInd]
   $exposures[$crossInd[$loopInd]]
```

are all legal, as long as 'crossInd' is itself an array with enough elements.

The number of elements can be obtained by using '$#' instead of '$' in front of the array name, such as '$#exposures'.

An array can be made larger by using it in an array assignment with additional values. Thus:

```
    exposures = {$exposures ($base + 1.5)}
```

would add one element to the array. However, this method is not restricted to add one element at the end; it can be used to add multiple elements or add elements at the front of the array. If the value expression contains other arrays, they will be added in their entirety, thus allowing arrays to be concatenated. An expression like '$exposures', i.e., an array variable not followed by an index, should be used only in assignments to new arrays and in the Echo command. The value of this expression actually consists of the elements separated by newlines, which are turned into spaces by Echo but make the variable not useful in other contexts.

Once an array is defined, individual elements can be assigned to by placing an index after the variable name. This index can be a constant, a variable, or an indexed array variable. For example

```
    exposures[2] = 4
    exposures[$loopInd] = $base
    exposures[$crossInd[$loopInd]] = 1.5
```

are all legal as long as the variables have appropriate values. Finally, the item being assigned to one array element can itself be an array variable or an array expression in '{ }'; the sequence of elements will replace the one being assigned to and the array size will increase accordingly.

If you use arrays and there is any possibility that the script will be distributed and possibly run on earlier versions of SerialEM, you should include the command

```
   Require arrays
```

somewhere in your script to keep it from being used on versions that do not support arrays.

Arithmetic Expressions

An arithmetic expression can contain numbers, variables that have numeric values, the 4 operators '+', '-', '*', and '/', and some arithmetic function names. Parentheses can also be included for clarity and to control the order of evaluation. Expressions within parentheses are evaluated separately and replaced by a single number, working outwards from the deepest set of parentheses. An expression is scanned first from left to right for "*' and '/' together; the number to the left and right of an operator are replaced by the result. Then the expression is scanned from left to right for '+' and '-' together and those operators are applied. Finally, it is scanned for some arithmetic function names. The table below lists the functions that take one or two arguments, which are just the following one or numbers (each possibly derived from arithmetic operations). Without parentheses, this order of evaluation means that functions will only work at the beginning of an expression, and that the function will be taken of the whole remainder of the expression. Technically, the solution to this restriction is to enclose the function and its arguments in parentheses, such as

```
   (ATAN2 $x +1 $y / 2)
```

but the program will also accept the more conventional placement of arguments within parentheses, as in

```
   ATAN2 ($x + 1 %y / 2)
```

There must be a space between the function name and the opening parenthesis. RAND is also available to obtain a random number between 0 and 1; it takes no arguments and either has to be assigned to a variable or written as (RAND) to use in any expressions. Arithmetic can be done only in variable assignment statements and in IF or ELSEIF statements.

Parentheses may be surrounded by spaces or attached to the items that they enclose. Specifically, one or more left parentheses can be at the beginning of an item, and one or more right parentheses can be at the end of an item. For example

```
    SIN_((12_+_$var)_/_2) is equivalent to SIN_(_(_12_+_$var_)_/_2)
```

where the underscores represent spaces. Other than this flexibility, there must be a space between each value and each operator.

Arithmetic functions with one argument	
SQRT	Square root of following number; generates error if it is negative
SIN	Sine of following angle in degrees
COS	Cosine of following angle in degrees
TAN	Tangent of following angle in degrees
ATAN	Arc-tangent of following number, between -90 and 90 degrees
ABS	Absolute value of following number
NEARINT	Nearest integer to following number
Arithmetic functions with two arguments	
ATAN2	Arc-tangent of first number divided by second number, between -180 and 180 degrees
MODULO	Remainder upon dividing nearest integer to first number by nearest integer to second one
POWER	First following number raised to the power of the second number
ROUND	Round first following number to the number of decimal places given by second number
DIFFABS	Absolute value of difference between the two numbers
FRACDIFF	Fractional difference between two numbers: absolute value of difference divided

by maximum of the two absolute values

IF, ELSEIF, ELSE, ENDIF, BREAK and CONTINUE Statements

An IF statement is used to start a block ending in ENDIF, in which statements are executed conditionally on the truth of the expression in the IF statement. The IF statement can contain one or more comparisons between two numbers, joined by the logical operators AND or OR. Each numerical comparison is referred to as a 'clause' (including in error messages). The format of the IF statement is thus:

```
if clause1 logical_operator clause2 logical_operator clause3 ...
```

where each clause consists of:

```
expression1 comparison_operator expression2
```

Here, each expression may be an arithmetic expression with multiple components, as long as it evaluates to a single number. The comparison operator occurs between two values and is one of '<', '>', '<=', '>=', '==', and '!=' (the latter two test for equality and non-equality). If the statement is true, then following lines are executed until an ELSE or ELSEIF is encountered, if any, at which point lines are skipped until the ENDIF. If the statement is false, following lines are skipped until either an ELSEIF, an ELSE, or an ENDIF is encountered.

The format of the logical expression in an ELSEIF statement is the same as that in an IF statement. When an ELSEIF is encountered and no previous test in the IF block has been satisfied, then the expression is evaluated and the same actions are taken as for an IF statement.

Just as for an arithmetic expression, parentheses are allowed for grouping the logical expressions and controlling how they are evaluated. The deepest set of parentheses containing a logical operator is evaluated first and replaced by a simple true or false clause, and evaluation works outward from there. Within parentheses or in their absence, a logical expression is evaluated from left to right. If there are more than two clauses, a cumulative truth value on the left is combined with the truth value on the right for each logical operator. For example, for

```
$A < $B OR $A < 12 AND $I == 5
```

the truth of whether A < B or A < 12 is determined and ANDed with whether I is 5. As programmers have learned over the years, relying on the order of evaluation in complex expressions is prone to error, so just use parentheses.

```
($A < $B OR $A < 12) AND $I == 5
```

makes it clear what is happening, and

```
$A < $B OR ($A < 12 AND $I == 5)
```

gives the different result that a person used to AND having precedence over OR would expect.

A common use of IF statements would be for loop control using the BREAK, CONTINUE, or SKIPTO statements. BREAK causes termination of the innermost loop being executed; the script then continues with statements after the next ENDLOOP. CONTINUE causes statements to be skipped to the end of the innermost loop, but the next iteration of the loop is then run, if any.

Labels and SKIPTO Statements

The SKIPTO statement can be used to jump ahead to a location defined by a label. A label is the only item on a line and ends in a colon; any character string not starting with $ is allowed. The SKIPTO statement includes the label without the colon. For example,

```
if $error > 0
  SkipTo Cleanup
endif
......
Cleanup:
......
```

The SKIPTO can be used to jump forward within a loop or IF block, but it cannot be used to jump into a different loop, a different IF block, or even a different ELSE/ELSEIF section of the same IF block. SKIPTO statements would be most useful for breaking out of multiple loops at once, for skipping over substantial numbers of lines where an IF statement does not seem suitable. SKIPTO is like GOTO in older programming languages, which is disapproved of for good reasons. It is thus recommended that you use it only when other control constructs would be clumsy and result in a script that is harder to follow. For example, if you have commands to restore initial settings that must be run from multiple places before exiting, it is preferable to put those in a function instead of using SKIPTO.

Defining and Calling Functions

It is possible to define named functions within scripts. These functions must begin with *Function*, end with *EndFunction*, and occur after any non-function commands in a script. (Commands after an *EndFunction* will not be run). The Function statement includes the name of the function, which must be all one word without spaces, two optional entries for the number of numeric arguments and whether there is a string argument, and optional variable names for arguments to be assigned to.

```
 Function FunctionName #_of_numeric_args 1_if_string_arg argName1 argName2 ...
```

Any integer other than a 0 in the last position indicates that there is a string argument. If any variable names are entered, they must be preceded by both of the numeric values. There need not be as many variable names as arguments.

Like script names, function names are case sensitive. There is no limit to the number of functions in a single script, although they must all have distinct names. The same name can be used for functions in different scripts; however, if this is the case, functions must be called with the script name or number as shown next.

A function is called with *CallFunction*, followed by the name of the function, the numeric arguments if any, then the string argument if any. The string argument can include spaces; all text after the numeric argument is used for the string argument. For example, for a function defined as

```
Function AlignWithBuffer 1 1
```

the call could be

```
CallFunction AlignWithBuffer $ifNeedImage $buffer
```

where the two variables would indicate whether a new image is needed and what buffer letter to align with. If AlignWithBuffer occurs in more than one script, and the one in script 15 whose name is 'Many Funcs' is intended, the call would need to be either

```
CallFunction 15::AlignWithBuffer $ifNeedImage $buffer
```

or

```
    CallFunction Many Funcs::AlignWithBuffer $ifNeedImage $buffer
```

Notice that it can handle spaces in the script name but not the function name. However, as of SerialEM 3.7 (May 17, 2017), if a function occurs in the current script as well as in other ones, the function in the current script will be used when there is no script number or name in front of the function name.

Inside the function, the arguments are defined as regular variables: either the variables listed at the start of the function, or 'argVal1', 'argVal2', etc. If you listed fewer variable names than arguments, the last arguments will be assigned to 'argVal' variables with the same numbers as they would have had if no variables were listed. The arguments are actually optional when calling a function, so if a numeric argument is not supplied in the call, the corresponding variable is 0, and a string variable will be empty if the string argument is not supplied. The number of arguments actually supplied in the call is assigned to the variable 'numCallArgs'. As of SerialEM 3.7.0 beta, this variable and the arguments are all created as local variables, which means they are not accessible if this function calls another script or function (which gets its own argument variables), they will not be lost if the function calls another function, and they become undefined when returning from a function. As mentioned above, all other variables are global by default: variables defined by the calling script are available within a called function or script, and variables defined within a function or script are retained when it returns.

If you list variables to be assigned in function definitions and there is any possibility that the script will be distributed and possibly run on earlier versions of SerialEM, you should include

```
   Require variables1
```

somewhere in your script. Earlier versions will not pay any attention to variables on a Function line and will not work correctly, so this command is good way to catch the problem.

If you have a cleanup function that restores initial settings, you can use the command 'OnStopCallFunc' to have the function called whenever there is an error or the user presses STOP. Ideally, this should be placed after commands that record the initial state; otherwise, the function should use 'IsVariableDefined' to test if particular variables are defined before using those variables.

The existence of functions presents new possibilities for errors, and some of these errors cannot be detected in the initial checking of scripts. For example, a function that has already been called cannot be called again before it returns, this error may not be detected until the second call of the function during execution.

Advanced Examples

This script will save the current magnification and change to 20000x, then move the stage to a 4 by 4 array of positions centered around the current location and autofocus, take a Record, and save it at each position. Then it will restore the mag and the stage position. This is done in a function that is set up to be called if the user stops the script. The indentation is for readability only.

```
# A script to illustrate variables, loops, and arithmetic

MacroName Example
Require variables1
```

```
ReportMag oldMag
ReportStageXYZ baseX baseY
OnStopCallFunc RestoreState
SetMag 20000
Loop 4 ix
   x = $baseX + $ix * 3 - 7.5
   Echo Doing column at X = $X
   Loop 4 iy
      y = $baseY + $iy * 2 - 5
      MoveStageTo $x $y
      Autofocus
      Record
      Save
   EndLoop
EndLoop
CallFunction RestoreState

Function RestoreState
MoveStage $baseX $baseY
SetMag $oldmag
EndFunction
```

This example will take pictures at a certain time interval, measure the displacement between each pair of pictures, and do this repeatedly until the displacement falls below a criterion or the limit on the number of iterations is reached. Note that important parameters are defined as variables at the top of the script, so it would be easy for users to set parameters without having to be adept at writing such a script.

```
# A script to take a picture after drift falls below a criterion
MacroName Drift
shot = Trial
interval = 5
times = 4
crit = 0.7
SuppressReports
$shot
Delay $interval
Loop $times index
   $shot
   AlignTo B
   ReportAlignShift
   ClearAlignment
   dx = $reportedValue3
   dy = $reportedValue4
   dist = sqrt ($dx * $dx + $dy * $dy)
   echo Distance = $dist nm
   if sqrt ($dx * $dx + $dy * $dy) < $crit
```

```
        echo Drift is low enough after shot $index
        break
    endif
    if $index < $times
        Delay $interval
    else
        Pause Drift never got below $crit: Continue anyway?
    endif
EndLoop
Record
```