MortScript V4.0

(c) Mirko Schenk
mort@sto-helit.de
http://www.sto-helit.de

Contents

1 What is MortScript? / License

MortScript is "just" an interpreter (similar to the runtime environment of Visual Basic), so there's no visible program that can be run for itself. (Except for the registration of file extensions when you run MortScript.exe, but that's unnecessary if you used an installer.)

You have to use downloaded .mscr or .mortrun script files, or write them yourself with any text editor (plain text, no Unicode!). For beginners, writing own scripts might be a bit complicated.

You can execute those scripts by running them like any other application in the file explorer (i.e., just tap the file), or create a shortcut to those files in your start menu (\Windows\Start menu, or some localization) with the file explorer (or any other tool).

I don't offer any warranty for damages caused by this program (neither me nor the script authors are perfect...). Be aware that foreign scripts can do lots of dangerous things, just like any "normal" application can read or delete files, or send data via the internet.

MortScript is a script language focused on batch control, i.e. to run other applications and remote control them, and to do basic system operations, like file operations, registry modifications, and such. Due to this, only basic dialogs are available.

MortScript is freeware, i.e., you don't need to pay for it. However, modifications are not allowed.

You may deploy MortScript with your own scripts, even for commercial scripts (be aware they will be available in source code since there's no compiled code...). But you have to note at a sensible location (readme.txt, setup, or similar) that MortScript is a foreign product with its own license. A link to my web site or mentioning my name would be good manner (e.g. „MortScript is freeware, www.sto-helit.de“).

I'd be glad about a small (or big ;)) donation as "thank you!". See also 11 Donations

2 Functional range

MortScript supports among others.:

- Run, activate, hide, and close applications
- Wait functions: certain timespan, wait for existence or activation of windows
- Send keystrokes and mouse clicks to windows
- Copy, rename, move, delete files, create shortcuts
- Create and remove directories
- Supports ZIP archives (no overwriting of contained files!)
- Read and write text files
- Read and modify the registry
- Internet: Reading text files, downloads, create and close connections
- If conditions, Choice selections and ForEach, While or Repeat loops
- Some system features (e.g. rotation, volume, backlight brightness, soft reset)

3 Installation

3.1 Different MortScript variations

MortScript is available for PCs, PocketPCs, Smartphones (with Windows Mobile) and PNAs (Navigation systems based on Windows Mobile). The functional range variates depending on the possibilities of the devices. If a function doesn't exist for a certain variation, it's noted in its description in this manual. It's also possible to check which MortScript variation is used (see 9.25.1 Indentify MortScript variation (MortScriptType)).

The installation downloads contain all variations. You have to select the one that fits your system. The system is abbreviated this way:

PC = PC (Windows XP/Vista)
PPC = PocketPC
SP = Smartphone
PNA = Navigation device

3.2 PC Setup

Just execute the MortScript-4.0-*system*.exe (e.g. MortScript-4.0-PPC.exe) from the “exe” directory in the archive on your desktop PC and follow the directives...

Currently, there's no setup for the PC variation, please see “Binaries” for that...

3.3 CAB file

Copy MortScript-4.0-*system*.cab from the “cab” directory in the archive to the target device (use “Browse” in ActiveSync or use a storage card) and open it with the file explorer of the device (or any alternative file manager, like TotalCommander, Resco Explorer, and such).

This installation type is only available for Windows Mobile, so there isn't a CAB file for the PC version.

3.4 Binaries

Copy the contained files from the archive's subdirectory of “bin” that's named after the desired device type (e.g. “bin/PPC”) to any place on the device (e.g. to “\Program files\MortScript”). Then run MortScript.exe there, so the required registry entries will be created (for the assignments to the script extensions .mscr and .mortrun).

4 Usage

4.1 Create and execute scripts

MortScript executes files with the extensions ”.mscr” and ”.mortrun”.
The latter for backward compatibility, the program formerly was named "MortRunner".

Such a file can be created with any text editor. You can even use PocketWord, but you have to use "Save as - Text" and rename the extension from .txt in .mscr or .mortrun afterwards. If your editor supports multiple formats, please use “ANSI”.

If this file is opened – e.g. by tapping it in the file explorer – the lines in this file will be executed sequentially - just like a batch file.

You can create a link to the file you created in the start menu or autostart folder. You can do this in the file explorer by “copying” the file, and “pasting a shortcut” in ”\Windows\Start Menu” or ”\Windows\StartUp” (might be localized on your device).

4.2 Parameters for MortScript.exe

The parameter for MortScript.exe is the script file that should be executed, with the entire path. If it contains spaces, this argument must be surrounded by quotes (e.g. "Storage Card\myscript.mscr").

The PPC variation has an additional optional parameter /wait=*n*, whereby *n* is the number of seconds, which MortScript will wait for the existence of the given script file. This option is available, because the storage cards aren't available directly after resuming from standby. So if you assigned a script to an application button, and wake up the device with it, it might happen the script can't be opened. By default, MortScript will wait for 5 seconds.

Additionally, all parameters in the format “name=value” are set as variables to use in the script. For example, if “test="This is a test"” is given as parameters, “Message(test)” in the script would show “This is a test”.

4.3 Multiple instances and aborting scripts

MortScript can run in multiple instances, but only once for each script.

If an already running script is run a second time, an open dialog of that script (e.g. Choice, Message, ...) will be activated. If the script doesn't show any windows, nothing will happen.

If you want to terminate running scripts, you can use ScriptProcExists and KillScript. See also informations in 9.20.5 End a running script (KillScript).

5 Additional tools

5.1 Execute scripts when a storage card is inserted or removed

Autorun.exe allows you to use Windows Mobile's autorun feature in a quite easy way.

If a storage card is inserted or removed, Windows executes the program “autorun.exe” in the folder “2577” (CE code for ARM processors) or “0” on that storage card (e.g. ”\Storage\2577\autorun.exe”).
This feature isn't supported on all devices. For example, I've read that HP deactivated this feature on iPAQ 2210. It also doesn't work on PNAs and PCs, these autorun.exe versions are only for use as “dummy exe” (see below).

If you copy autorun.exe, MortScript.exe as well as autorun.mscr and/or autoexit.mscr to this folder, the scripts autorun.mscr (after inserting) and autoexit.mscr (after removing) will be executed (if the corresponding script exists).

For backward compatibility, you can also use autorun.mortrun and autoexit.mortrun.
If both .mscr and .mortrun do exist, the .mscr is used.

5.2 “Dummy exe” for scripts

If you rename autorun.exe, it will execute the fitting script. I.e., if autorun.exe is renamed e.g. to myscript.exe, it will execute myscript.mscr, or, if it doesn't exist, myscript.mortrun.
The renamed autorun.exe and script must be located in the same directory.

If a MortScript.exe is located in the same directory, too, it will be used to execute the script, otherwise the default assignment is used (i.e. the installed MortScript – or you'll get an error message if there's no installation...).

This feature is handy for programs which can run other programs but not do not allow to select .mscr files, like some phone profile tools for the PhoneEdition devices.

5.3 Supporting scripts for CAB installations (setup.dll)

setup.dll is only available for PocketPCs. It allows to execute scripts automatically after installation or before deinstallation when a CAB installation is used. This way, it saves developers to create an own setup DLL. If CAB files are a closed book to you, just skip this chapter... ;-)

To activate the setup.dll, it must be set as setup DLL of the CAB file. If you're using the CAB wizard from Microsoft, this is done with `CESetupDLL = "setup.dll"` in the .inf file, other tool should offer a menu entry or option for it.

MortScript.exe and – if you use ZIP archives – mortzip.dll must be installed to the default installation directory (%InstallDir%) by the CAB installation. Same goes for install.mscr (executed after installation) and uninstall.mscr (executed before deinstallation), whereby those two can be omitted. But of course that only makes sense for one of them, cause otherwise the setup.dll wouldn't do anything...

6 Important general informations

6.1 Glossary

Constant: A fixed value, i.e. numbers like “100” or strings like "Test"

Variable: A string that identifies an assigned value.
E.g. “x = 5” (“5” is assigned to variable “x”) or “Message(x)” (The value “5”, which is assigned to variable “x”, will be displayed).

Expression: A combination of variables, constants, functions (see below) and operators, which results in a single value (e.g. “5*x” or ”"Script type: " & ScriptType()”)

Assignment: Setting a value to a variable, usually done with “v*ariable name* = e*xpression*“

Parameter: Expression results which are passed to commands or functions

Command: An instruction without a return value, e.g. MouseClick or Message

Function: An instruction which returns a value, e.g. SubStr. It's used in expressions, so it can only be used in assignments or parameters.

Control structure: Instructions, which modify the course of the script, like If, Choice, ...

6.2 Syntax style in this manual

The style in this manual is loosely based on the (E)BNF:

`bold`:	Fixed value, e.g. the command or function name
`italics`:	Variable value, usually any expression
`[...]`:	Optional, can be omitted (usually, default values will be used in this case)
`{...}`:	Can be repeated or omitted
`x|y|z`:	Either x, y, or z must be used (usually fixed values).
`(...)`:	Grouping (usually to clarify "|" options).

If the characters are bold, they must be entered that way, e.g. parentheses (**`(`**...**`)`**).

Generally, this syntax is used:
`Command` `[` **`(`** *`Expression`* `{`**`,`** `Expression }` **`)`** `]`
or
*`Variable = Function`***`(`** `[` *`Expression`* `{`**`,`** *`Expression`* `} ]` **`)`**
whereby an single function call is just a special type of an expression. For more about expressions, see 7 Possible parameters and assignments.

When using one of the (few) commands, which don't require any parameters, the parentheses after it are optional, i.e. it's up to your liking whether you write e.g. „RedrawToday“ or „RedrawToday()”. But this is NOT that way for function calls! (That's because in expressions, the parentheses define that the name in front of them defines a function call, otherwise it would be a variable name.)

6.3 Spaces, tabulators, and line breaks

Spaces and tabulators are allowed at any location before, after, and between the single elements (i.e. around command/function names, parentheses, values, operators, ...). In quoted strings, there're a part of the string, otherwise they're ignored.

Line breaks within an instruction are possible, if you put a ”\” at the end of the line that should be continued. You can do this inside of strings, too. But all surrounding spaces, tabulators, and the line break itself will be replaced with a single space.

Example:

```
Message( "This is \
         a test" )
```

will show „This is a test“.

6.4 Case sensitivity

Commands and file names are not case sensitive, but window titles are. But you can use only parts of the window title. I.e. `Show("WORD")` will not work, but `Show("Word")` will also activate "Pocket Word" (or the first window with "Word" in the title...)

6.5 Directories and files

Directories and file names must always be given with complete path (e.g. `"\path\to\file.ext"` or `"\some\directory"`), because Windows Mobile doesn't know a “current directory”.

6.6 Comments

Comments are possible by writing the character ”#” at the beginning of the line. Spaces in front of the ”#” are allowed.

7 Supported parameters and assignments

7.1 Expressions

All parameters for functions and commands, conditions for control structures, and values assigned with ”=” are expressions. (Exception: Old syntax, see 10 Old syntax and commands)

Expressions consist of the following parts:
Fixed strings in quotes, e.g. `"Text"`
Fixed numbers, e.g. 42
Variables, e.g. x
Functions, e.g. SubStr(*parameter*)
Operators, e.g. +, -, &, ..., which define, how values (constants, variable contents, function results) should be combined.

That sounds more complicated than it is. A few examples will clarify that:

```
Message( "Hallo!" )
```
→ The command “Message” is invoked with the fixed string “Hallo!” as parameter

```
Sleep( 500 )
```
→ The command “Sleep” is invoked with the fixed number “500” as parameter

```
Sleep( pause * 100 )
```
→ In this case, the contents of the variable “pause” is combined with the fixed number “100” by the operator ”*” (multiplication).

```
Message( "Battery level: " & BatteryPercentage() & "%" )
```
→ Concatenates the two fixed strings with the result of the function “BatteryLevel()” and passes this value as parameter for “Message”.

```
message = "Battery level: " & BatteryPercentage() & "%"
```
→ Like above, but the result is assinged to the variable “message” instead of being a command parameter.

```
If ( BatteryPercentage() > 20 )
  # instructions
EndIf
```
→ An expression as condition

More informations about how constants and variables must be written, and which operators are available, follows in the next few chapters.
The available functions are documented in 9 Commands and functions.

7.2 Data types

MortScript doesn't know “typing”. This means, numbers and strings are automatically converted to each other when necessary.

Whether a value is interpreted as number or string, depends on the context.

For numerical operations (e.g. ”+”, more in operators), the values will be converted to numbers, if necessary. This means, ”"5"+"10"” would return 15. If a string doesn't contain a valid number, “0” (zero) is used.

The other way around, when a text operator is used (e.g. ”&”, which concatenates strings), numbers will be converted to strings, so „5 & 10“ will return „510“.

It's similar with parameters. Usually, the meaning of the parameter will decide whether it's used as number or string. For example, a text output is a string, while a timespan is a number.

For conditions and “on/off” parameters, there's the following rule: If the value represents a valid number except “0”, this means “condition fulfilled” resp. “on”, otherwise it's “not fulfilled” / “off”. I.e. expressions like 5, "10", 1=1, etc. are “true/on”, while 0, "x", 2=1 are “false/off”.

7.3 Fixed strings

Fixed strings must be surrounded by quotes (").
To use quotes inside a quoted string, you have to double them, e.g..

```
Message( "He said: ""This is a test""" )
```
→ Will show „He said: "This is a test"”.

The following combinations will be replaced with special characters:

^CR^ → Carriage Return
^LF^ → Line Feed
^NL^ → Windows-/DOS line bread in files (New line, consists of CR+LF)
^TAB^ → Tabulator

In Windows, a new line in text files usually is the combination of carriage return and line feed (^CR^^LF^ = ^NL^). But sometimes there are also files in Unix style, which use only ^LF^.

7.4 Fixed numbers

Numbers can simply be written as such (i.e. `x = 5`, `Sleep(20)`, ...).
MortScript doesn't support floating point operations, so no decimal points are supported!

7.5 Variables

Variables are „placeholders“ for a value which is assigned to them.

All parts of an expression, which are neither constants (e.g. 123 or "string"), operators (+, -, &, ...), nor functions (followed by parentheses) are interpreted as variable name.

Valid characters for variable names are the letters A-Z (no umlauts, accents, or similar!), digits, and underline (”_″). Variable names are not case sensitive, i.e. `MYVARIABLE` and `myvariable` are the same value.

A variable name mustn't start with a digit, because those digit(s) would be interpreted as a fixed number in expressions (and everything after it as operator or something invalid). So “9mod2” is the same as “9 mod 2”, and not a variable!

Assigning a variable usually is done with ”=”, e.g.

```
myvar = 5 * x + y
```

But there also are some commands and control structures which assign variables, like GetTime or ForEach.

To use a variable in expression, you just have to write its name, like the “x” in the example above.

Some variables are predefined, to allow better readable instructions. In spite to other languages, you are able to modify them, but you shouldn't do that:
TRUE, ON, YES are initialized with “1”,
FALSE, OFF, NO are initialized with “0”,
CANCEL is initialized with “2”.

If you use the old syntax, be aware the usage of variables was a bit more complicated back then (%...% etc.).

7.6 Arrays (Lists)

Arrays are a special type of variables. An array consists of multiple variables which belong together, so called elements.

An element is addressed with the variable name and the so called “index” in brackets. This means, array[1] identifies the element “1” of the array “array”.

It's also possible to use strings as index, e.g. colors["blue"], whereby the index, like the variable name, is not case sensitive. I.e., COLORS["BLUE"] identifies the same element as colors["blue"].

For both assignment and in expressions you can use any expression for the index. That's the main advantage of arrays, because usually the elements are accessed with a variable as index (e.g. some counter variable).

Some instructions (like Choice or Split) support array parameters. But they'll only access the elements with the indexes from 1 to the first not assigned number. Smaller indexes (<= 0), elements after a gap, or with string indexes will be ignored.

Examples:

```
array[ "1" & 1 ] = "eleven"
Message( array[ (2-1) & "1" ] )

list[1] = "a"
list[2] = "b"
list[5] = "f"
list["a"] = "A"
idx = Choice( "Selection", "Choose something", 0, 0, list )
```

→ Only “a” and “b” will be shown in the choice dialog.

7.7 Operators

7.7.1 List of all possible operators

All possible operators by priority (highest first):

`()`	Parentheses
`NOT`	Negation
`* / MOD`	Multiplication, division, modulo (remainder of divisions)
`+ -`	Addition, subtraction
`& \`	Concatenation of strings
`> >= < <= = <>`	Numerical comparisons
`gt ge lt le eq ne`	Alphanumerical comparisons
`AND &&`	Binary / logical “and”
`OR ||`	Binary / logical “or”

7.7.2 Logical and binary operators

For logical operations (true or false, i.e. `&&`, `||` and `NOT`) there's the following rule: If the value represents a valid number except “0”, this means “condition fulfilled” resp. “on”, otherwise it's “not fulfilled” / “off”.
I.e. expressions like 5, "10", 1=1, etc. are “true/on”, while 0, "x", 2=1 are “false/off”.
„NOT 5“ would return „0“ (something not 0 = true will become false = 0), „NOT (2-2)” will be „1“ (2-2 = 0 = false becomes true = 1).

The difference between `AND` and `&&` resp. `OR` and `||` is that for `&&` and `||` every value which isn't 0 is handled like 1 (because 2 is as “true” as 1). If you use those operators only to combine the results of comparisons or check functions, there'll be no difference, because they'll only return 1 (true) and 0 (false) anyway.

The binary operators `AND` and `OR` additionally are useful for are bitwise checks, e.g. ”`(x AND 4) = 4`” will check whether the 3rd bit (4 = binary 100), is set in the value of variable “x”.

The logical operators `&&` and `||` are primarily thought for “C hackers”, which are used that 1 AND 2 is not 0 (binary 01 AND 10 would result in 0) but 1 = “true”.

7.7.3 Comparisons

Numerical and alphanumerical comparisons have the same priority, they've been split in the operator list only for better overview.

Since MortScript doesn't support typing, the operator has to decide whether the values are compared as numbers or as strings. This means, `"123" < "20"` is “false” (because 20 is smaller than 123), but `123 lt 20` is “true” (because the character “1” is smaller than “2”, just like “a” is smaller than “b”).

If you can't memorize the alphanumerical operators: they're just the abbreviations of „greater than“, „greater/equal“, „less than“, ”(not) equals“, etc.

7.7.4 Concatenation of strings and paths

”\” is an operator for the concatenation of paths. There'll be only one ”\” at the concatenation point. In spite to this, ”&” simply concatenates the strings, which might result in invalid paths.

Example:
```
"\My documents\" \ "\file.txt"
"\My documents" \ "file.txt"
"\My documents\" \ "file.txt"
```
→ will all result in `"\My documents\file.txt"`.

In spite to this:
```
"\My documents\" & "\file.txt"
```
→ "\My documents\\file.txt"
```
"\My documents" & "file.txt"
```
→ "\My documentsfile.txt"
```
"\My documents\" & "file.txt"
```
→ "\My documents\file.txt" – the only valid file name!

8 Control structures

8.1 Conditions

As condition, any expression in parentheses can be used. The condition is fulfilled, if its result (if necessary, after converting to a number) is not 0 (zero).
Possible functions are listed in the fitting category of 9 Commands and functions. Read also the chapter 7, esp. 7.2 Data types for further informations.

Examples:

```
If ( wndExists( "Word" ) )
EndIf

While ( x <> 5 )
EndWhile
```

8.2 Simple branchings (If)

```
If( expression )
  { instructions }
[ Else
  { instructions } ]
EndIf
```

Executes the lines between If and Else or EndIf, if the condition is fulfilled, or the lines between Else and EndIf (if Else exists), if it's not.
If, Else and EndIf each must be in a separate line.

8.3 Branching by values (Switch)

```
Switch( expression )
Case( value {, value } )
  { instructions }
{ Case( value {, value } )
  { instructions } }
EndSwitch
```

Depending on the result of the expression, the blocks which have listed the value in the “Case” line are executed.

The values can appear in multiple case blocks (e.g. „Case(1, 2)” and „Case(2, 3)”). The “fitting” Blocks will then be executed in the order of appearance.

“Slipping through” or “break” like in C is not possible, but similar things can be done much more clearly arranged by using a value in multiple “Case”s.

Only numerical comparisons are supported.

8.4 Branching with selection dialog (Choice, ChoiceDefault)

```
( Choice( title, hint, value, value {, value } )
| Choice( title, hint, array )
| ChoiceDefault( title, hint, default, timeout, value, value
                 {, value } )
| ChoiceDefault( title, hint, default, timeout, array )
)
Case( value {, value } )
  { instructions }
{ Case( value {, value } )
  { instructions } }
EndChoice
```

Shows a selection with the given values. At "Case", you have to use the number of the entry (starting with 1). Pressing “Cancel” or no selection will return 0.
Apart from that, it works similar to “Switch”.
In theory, you could also use `Switch( Choice( ... ) )` (Choice as function, see 9.19.5 Selection from a list (Choice)), but Choice as control structure looks better.

ChoiceDefault is a variation which enables to set a default selection and a timeout after which the selected entry is used. If the user selects another entry, the countdown will be restarted.
The *default* value must be given as index (i.e. “2” for the 2nd entry).
0 is allowed for no default (i.e. "Cancel" if the user doesn't select an entry).
The *timeout* is given in seconds. With 0, no timeout will be used.

Example:

```
Choice( "Test","Select a number","One","Two","Three" )
Case( 1 )
  Message( "One" )
Case( 2, 3 )
  Message( "Two or three" )
Case( 3 )
  Message( "Three" )
Case( 0 )
  Message( "Cancel" )
  Exit
EndChoice
```

8.5 Conditional loop (While)

While(*condition* **)**
    { *instructions* }
EndWhile

Executes the lines between While and EndWhile as long as the condition is fulfulled.
While and EndWhile must be in separate lines.

8.6 Iteration over multiple values (ForEach)

ForEach *variable*{, *variable* } **in** *type* **(** *parameter* {, *parameter* } **)**
  { *instructions* }
EndForEach

This is quite a mighty tool. The given variable(s) is/are set to the values defined by the type and parameters in each iteration. This varies from simple value lists (type “values”) to keys and values of sections in INI files (“iniKeys”).

Please note: If an array element is used as iterator variable, its index will only be parsed when the loop is entered. This means if “i” is 1 when the ForEach loop is entered, “array[i]” will assign “array[1]” in every iteration, no matter whether “i” is assigned another value in the loop block! Same goes for the parameters: They're evaluated when the loop is entered.

If not told otherwise, you can use expressions for all parameters, just like with almost every other commands.

Currently, there are the following variations:

ForEach *variable* **in values (** *value* {, *value* } **)**

Assigns each of the given values to the variable in each iteration.

ForEach *variable* **in array (** *array variable* **)**

Assigns the element values of the array to the variable. Only elements from 1 to the first number that isn't assigned as index are regarded. Lower values, alphanumerical indexes, and values after a gap are ignored.

ForEach *variable* **in split (** *string*, *separator*, *trim?* **)**

Similar to the Split function (see 9.3.1 Spit string to multiple variables/array elements (Split)), but the single parts are assigned to the variable one after the other.

ForEach *variable* **in charsOf (** *string* **)**

Assigns the variable each character of the string one after the other. Since MortScript automatically converts types, this also works for numbers – it'll assign e.g. “4” and “2” (for 42).

ForEach *variable* **in iniSections (** *file name* **)**

Returns the single sections ("[Section]", without brackets) of the given INI file.

ForEach *key*, *value* **in iniKeys (** *file name*, *section* **)**

Assigns the contents of a section in an INI file to the given variables.
„Key“ is the name of the variable that will receive the entry name in front of the ”=”, „value“ is the variable that receives the value after it.

ForEach *variable* **in files** (*search expression*)
ForEach *variable* **in directories** (*search expression*)

Assigns the found files resp. directories to the variable.
The expression must consist of a path and a filename expression with jokers, e.g. `"\Program Files\Mort*"` or `"\Program Files\Test\*.exe"` (similar to XCopy or Move).

8.7 Fixed number of repeatings (Repeat)

Repeat (*count* **)**
  { *instructions* }
EndRepeat

Repeats the commands between those two commands *count* times.
Count must be at least 1.

8.8 Sub routines (Sub / Call)

```
Sub subroutine name
  { instructions }
EndSub

Call subroutine name
```

With “Call”, the script will continue at the line following the “Sub” with the same parameter. When the end of the subroutine (EndSub) is reached, it returns to the line after “Call”.

In spite to most other instructions, the subroutine name must follow “Sub” without parentheses. Expressions are not allowed!

For “Call”, in theory the variation “Call(*expression*)” is possible, too. In this case, the expression must result in a valid subroutine name. But usually “Call mySubroutine” looks better than “Call("mySubroutine")” or even “Call(variableContainingMySubroutine)”. The latter might be used as some kind of “On ... gosub ...” (for those who remember that BASIC command), but it's not good programming style, and might cause troubles if you forget a possible subroutine.

There are no parameters or return values, all variables are global.

The subroutines must follow the main program, MortScript exits on the first “Sub” it encounters.

8.9 Other script as subroutine (CallScript)

```
CallScript( MortScript file )
```

Executes the given script as if it were a subroutine.
All variables remain global, i.e. the variables set in the current script are also available in the invoked script, and the modified and additionally set variables will remain in the calling script.
The script must be given with full path and file name.

Example:

```
CallScript( SystemPath("ScriptPath") \ "subscript.mscr" )
```

To execute other applications or „independent“ scripts, there are own commands. see 9.5 Execute applications or open documents.

8.10 Abort script (Exit)

Exit

Stops executing the script.

9 Commands and functions

Functions are represented by *x* `= Function( ... )`. Of course, they can be used in more complex expressions, too (see 7 Supported parameters and assignments).

9.1 Error handling (ErrorLevel)

ErrorLevel (*error level* **)**

Decides which error messages will be shown. The error level has to be a string (e.g. "syntax"), i.e. **not** “ErrorLevel(syntax)” - except „syntax“ would be a variable which contains "syntax"...
The default is „error“.

It also might change the program flow: If the errorlevel is "warn" or "error", the program will be terminated if an error event (see list below) occurs. If the level's "off" to "syntax", the error will be ignored, so you can check it e.g. with “If (wndExists(...))”.

Possible error levels:

off	**No error messages** The script might be terminated without any message
critical	**Critical messages** currently none, reserved for future use
syntax	**Syntax errors** e.g. wrong parameter count or invalid command or function names
error	**Other errors** e.g. nonexistent windows, troubles writing or deleting registry entries or files, a new document or directory couldn't be created, ...
warn	**Warnings** e.g. if a file/directory couldn't be removed, copy/move/rename didn't work (target already existing?)

The levels include all levels that are listed above, i.e. with “error” the messages of the levels “syntax” and “critical” are shown, too.

9.2 Assigning variables (”=” and Set)

Variable **=** *expression*

Assigns the result of the expression to the variable.

Set(*variable*, *expression* **)**

Assigns the result of the expression to the variable.
Shouldn't be used anymore, better use *Variable* = *expression*.
However, Set has a little “special feature”: If you give a variable enclosed by %...%, the contents of that variable is used as variable name. If e.g. “varRef” contains the string "var", and %varRef% is given as variable, the expression's result is assigned to the variable “var”, not “varRef”.
If „varRef“ contains a string enclosed with ”%”, this will even work recursive, until a variable contents without %s is found (or the system crashes with a stack overflow...)

9.3 String operations

9.3.1 Spit string to multiple variables/array elements (Split)

Split(*string*, *separator*, *trim?*, *variable* {, *variable* } **)**

Splits the string on each occurrence of the separator (single character), and assigns the parts the to given variables.

If there are more variables than parts, the remaining variables will be empty, if there are more parts than variables, they'll be ignored.

If "trim?" is 1, any spaces surrounding the parts will be removed.

If only a single variable is given, it will be interpreted as array name, i.e., the single parts are assigned to variable[1] to variable[*n*].

Examples:

```
Split( "a | b | c", "|", 1, a,b,c,d )
```

→ a="a", b="b", c="c", d=""

```
Split( "a\ b \c.def", "\", 0, a, b )
```

→ a="a", b=" b "

```
Split( "one, two, three", ",", 1, list )
```

→ list[1]="one", list[2]="two", list[3]="three"

9.3.2 Split string and return a single part (Part)

```
x = Part( string, separator, index [, trim?] )
```

Splits the string on each occurrence of the separator (single character), and returns the part with the given index (i.e., 2 for the 2nd part). You can use negative indexes, too. -1 is the last part, -2 the part before the last, and so on.

If "trim?" is 1 or omitted, any spaces surrounding the part will be removed.

Examples:

```
x = Part( "a | b | c", "|", 2 )
```

→ x = "b" (2nd part, spaces trimmed)

```
x = Part( "a\ b \ c.def", "\", -1, 0 )
```

→ x = " c.def" (last part, no trimming)

```
x = Part( "eins, zwei, drei", ",", 4 )
```

→ x = "" (empty string for not existing parts)

9.3.3 Get the length of a string (Length)

```
x = Length( string )
```

Returns the number of characters in a string.

Example:

```
x = Length( "This is a test" )
```

→ x = 14

9.3.4 Extract a range of characters from a string (SubStr)

```
x = SubStr( string, start [, length ] )
```

Returns „length“ characters starting with the „start“th character.
If length is omitted, or length is bigger than the remaining characters, everything from start to the end of the string is returned.
If the string is shorter than “start”, an empty string is returned.
You can also pass a negative value for “start”. In this case the last characters of the string are used, i.e. -2 is the character before the last. If the value's too big (longer than the string), the first character of the string is used.

Examples:

```
x = SubStr( "abcdef", 2, 3 )
```

→ x = "bcd"

```
x = SubStr( "asdf", -3 )
```

→ x = "sdf"

9.3.5 Find a string in another string (Find)

x = **Find(** *string to check*, *string to search* **)**

Returns the position of the first character where the searched string is found. If the searched string is not contained, it returns 0. This should be checked, to avoid invalid functions like SubStr with a start position of 0.

Examples:
```
x = Find( "abcdef", "cd" )
```
→ x = 3

```
x = Find( "abcdef", "CD" )
```
→ x = 0 (case sensitive!)

9.3.6 Find last occurrence of a character (ReverseFind)

x = **ReverseFind(** *string*, *character* **)**

Returns the position of the last occurrence of "character" in the "string". In spite to Find, only a single character is allowed.
If the character is not contained, the function returns 0.

Example:
```
x = ReverseFind( "abcba", "b" )
```
→ x = 4

9.3.7 Convert to upper / lower case (ToUpper/ToLower)

x = **ToUpper(** *string* **)**
x = **ToLower(** *string* **)**

Returns the given string converted to upper (ToUpper) resp. lower (ToLower) case.
If a variable is passed as parameter, its content will not be modified (in spite to the old commands MakeUpper/MakeLower).
Depending on your system and localization, "special characters" like "ä" or "è" will not be converted!

Examples:
```
x = ToUpper( "Abcba" )
```
→ x = "ABCBA"

```
x = ToLower( "AbcBA" )
```
→ x = "abcba"

9.4 Expressions in a string (Eval)

x = **Eval(** *string* **)**

Evaluates the expression contained in the string and returns its result.

Example:
```
x = Eval( "1+5*x" )
```
→ x = 26, if x was 5 before

9.5 Execute applications or open documents

9.5.1 Open application/document and continue script (Run)

Run(*application* [, *parameter*] **)**

Runs the application. The script continues while the application is loaded and executed.
Links (*.lnk), parameters, and documents are possible, too.
The complete path must be specified.

Examples:

```
Run( "\Windows\StartMenu\Messages.lnk" )
Run( "\Windows\PWord.exe", "\My documents\doc.psw" )
```

9.5.2 Open application/document and wait until it's finished (RunWait)

RunWait(*application* [, *parameter*] **)**

Like Run, but waits for the program to exit.
.lnk files won't work here.

Please note this will not have the desired effect if the program was already running. This is due to Windows' “reactivation” behavior: The program is executed a second time. This second instance looks for an already existing instance, and, if found, will activate it and exit itself. Thus, the script will continue after the second instance has finished, but the old instance is still running.

9.5.3 Other script as sub routine (CallScript)

CallScript(*MortScript file* **)**

Executes another script as if it were a subroutine.
See also 8.9 Other script as subroutine (CallScript)

9.5.4 Create new document / element (New)

New(*menu entry* **)**

Creates a new document (resp. appointment or similar).
The menu entry must be given exactly how it's shown in the “New” menu of the Today screen. Be aware this varies depending on the system's localization!

Sadly, this useful function isn't as easy to use since Windows Mobile 5. In this case, you'll either have to try it, or look in the registry in HKEY_LOCAL_MACHINE\Software\Microsoft\Shell\Extensions\NewMenu.

Not available for: PC

Example:

```
New( "Appointment" )
```

9.5.5 Execute application at a given time (RunAt)

```
RunAt( Unix timestamp, application [, parameter] )
RunAt( year, month, day, hour, minute
     , applikation [, parameter ] )
```

Runs the application at the given time. For this, the program is added to the so called “Notification Queue”. The PPC will wake up from standby if necessary.

The “Unix timestamp” is the time in seconds since 01/01/1970. This variant is interesting in combination with TimeStamp(), e.g. TimeStamp()+86400 for an execution in 24 hours (* 60 minutes * 60 seconds = 86400).

On many devices, MortScripts can't be executed directly. Instead, you have to invoke MortScript.exe with the script as parameter, e.g.

```
RunAt( starttime, SystemPath( "ScriptExe" ) \ "MortScript.exe", \
       """" & SystemPath( "ScriptPath" ) \ "notify.mscr" & """" )
```

Another problem: On many PPCs with WM5, the device wakes up and runs the program, but the display stays off, and the device goes back to standby shortly after the program was started. It often helps to invoke ToggleDisplay(ON) at the start of the scheduled script, if not, only a system update or registry hacks might help.

Not available for: PC

9.5.6 Execute application on each power on (RunOnPowerOn)

```
RunOnPowerOn( application [, parameter ] )
```

Executes a program every time the device is switched on. For this the program is added to the so called “Notification Queue”.
You should think twice about using this command, since for example there can be nagging error messages if the given program is deleted or moved.

Please regard the hints in RunAt about running scripts or WM5.

Not available for: PC

9.5.7 Remove application from „Notification Queue“

```
RemoveNotifications( application [, parameter] )
```

Removes the program from the “Notification Queue”, i.e., it will no more be executed automatically at given times (RunAt) or events (like RunOnPowerOn).
If there are multiple entries (e.g. multiple “RunAt”s), all of them will be removed.

If a parameter is given, it will be checked and only entries with the corresponding parameter will be removed. Otherwise, all entries with the program will be removed, no matter which parameter is entered in the notification. To remove only entries without a parameter, use an empty string ("") as paremter.

Not available for: PC

9.6 Application windows

9.6.1 Show and activate a window (Show)

Show(*window title* **)**

Activates the window with the given title.

9.6.2 Minimize/hide a window (Minimize)

Minimize(*window title* **)**

Minimizes (or, on Windows Mobil systems, rather hides) the window with the given title.

9.6.3 Close a window / end application (Close)

Close(*window title*)

Closes the window with the given title. If it's the main window of the application, the application usually is closed (exited), too. However, some rare programs ignore it.

9.6.4 Get the title of the currently active window (ActiveWindow)

x = **ActiveWindow()**

Returns the title of the currently active window.

9.6.5 Check whether a window is active (WndActive)

x = **WndActive(** *window title* **)**

Returns 1, if a window with the given title is active, otherwise 0.
The given text can appear anywhere in the window title, the comparison is case sensitive. I.e., WndActive("top") returns 1, if the Today screen is active (it's title is "Desktop"), but not if "Top program“ is active.

9.6.6 Check whether a window exists (WndExists)

x = **WndExists(** *window title* **)**

Similar to WndActive(), but also returns 1, if the window exists in background.

9.6.7 Wait until a window exists (WaitFor)

```
WaitFor( window title, seconds )
```

Waits (max. the given time) until the given window exists.

9.6.8 Wait until a window becomes active (WaitForActive)

```
WaitForActive( window title, seconds )
```

Waits (max. the given time) until the given window is active.

9.6.9 Get window title / element contents (WindowText)

```
x = WindowText( x, y )
```

Receives the window text of the element that's located at the given position. In most dialogs you can get labels, button labels, or contents of edit boxes that way. It doesn't work as intended for application drawn elements (like in most games) or e.g. list boxes. **In this cases, it'll usually return nothing (empty string) or the application's title.**

9.6.10 Send special commands (SendOK, SendCancel, SendYes, SendNo)

```
SendOK [ ( window title ) ]
SendCancel [ ( window title ) ]
SendYes [ ( window title ) ]
SendNo [ ( window title ) ]
```

With these commands, pressing the corresponding button is emulated.
If no window title is given, the currently active window is used.

There's no guarantee these commands will work with every program, because the programmers aren't forced to use the default signals.

9.7 Keystrokes

9.7.1 Sending strings (SendKeys)

SendKeys([*window title*,] *string* **)**

Sends the string as keystrokes to the given or currently active (if no window title was given) window.

Examples:
```
SendKeys( "My window", "Hi, how are you?" )
SendKeys( "Some text" )
```

9.7.2 Sending special characters (e.g. direction keys) (Send...)

Send*Special* [**(** *window title* [**,** *Ctrl?*, *Shift?*] **)**]

Activates the given window and sends the given special character. If no window title is given, the currently active window is used.
Ctrl? and *Shift?* are switches for the corresponding keys. If the parameter is "1", the key is pressed with the special character.

There are there special characters:
`CR`..... Carriage return
`Tab`..... Tabulator
`Esc`..... Escape
`Space`..... Space
`Backspace`..... Remove character left to the cursor ("←")
`Delete`..... Remove the character right to the cursor („Del")
`Insert`..... „Ins." (usually toggles between overwrite and insert mode)
`Up/Down/Left/Right`..... Direction pad to the corresponding direction
`Home`..... „Home", to the beginning of the line or document
`End`..... „Ende, the the end of the line or document
`PageUp/PageDown`..... Page up / down („Page ↑" / „Page ↓")
`LeftSoft/RightSoft`.... „Display buttons" on Smartphones and PPCs since WM5
`Win`..... „Windows" key on Smartphones and PPCs since WM5 (Start menu)
`Context`..... „Context menu" on PCs and Smartphones/PPCs since WM5

Examples:
```
SendCR( "ERROR" )
SendDown
SendHome( "",0,1 )
```
(highlight to beginning of line)

9.7.3 Copy screen contents to clipboard (Snapshot)

Snapshot [**(** *window title* **)**]

Activates the given window (if a parameter is passed) and copies the screen contents to the clipboard. ("Print screen" function of the system, might not work with every program).

9.7.4 Sending Ctrl+key (SendCtrlKey)

SendCtrlKey ([*window title*,] *key* **)**

Sends Ctrl + *key* to the current or given window.
E.g., `SendCtrlKey( "v" )` sends Ctrl+V (insert from clipboard) to the current window.
The key is not case sensitive, "v" and "V" will do the same.
The key must be exactly one character. Of course, this can be the result of an expression, too (like a variable).

9.8 Mouse clicks / tapping

9.8.1 Single click (MouseClick)

MouseClick([*window title*,] *x*, *y* **)**
RightMouseClick([*window title*,] *x*, *y* **)**
MiddleMouseClick([*window title*,] *x*, *y* **)**

Simulates a mouse click at the given position.
If a window is given, the position is relative to its upper left corner. If the window has a border (e.g. message boxes and questions), it is included.
If no window is given, the upper left corner of the display is 0,0.

The Right... and Middle... commands simulate mouse clicks with the right resp. middle mouse button, and are only available in the PC variant.

9.8.2 Double click (MouseDblClick)

MouseDblClick([*window title*,] *x*, *y* **)**
RightMouseDblClick([*window title*,] *x*, *y* **)**
MiddleMouseDblClick([*window title*,] *x*, *y* **)**

Just like MouseClick, but sending a double click.
The Right... and Middle... commands simulate mouse clicks with the right resp. middle mouse button, and are only available in the PC variant.

9.8.3 Press / release the mouse button separated (MouseDown/MouseUp)

MouseDown([*window title*,] *x*, *y* **)**
MouseUp([*window title*,] *x*, *y* **)**
RightMouseDown([*window title*,] *x*, *y* **)**
RightMouseUp([*window title*,] *x*, *y* **)**
MiddleMouseDown([*window title*,] *x*, *y* **)**
MiddleMouseUp([*window title*,] *x*, *y* **)**

Simulates pressing resp. releasing the mouse button. The parameters are as those of MouseClick. These two commands should be used together. With those, you can simulate “Tap&Hold” (Sleep between them) or “Drag&Drop” (MouseUp on another position).

The Right... and Middle... commands simulate mouse clicks with the right resp. middle mouse button, and are only available in the PC variant.

9.9 Waiting

9.9.1 Fixed delay in milliseconds (Sleep)

```
Sleep( milliseconds )
```

Waits the specified time.

9.9.2 Wait message with countdown / condition (SleepMessage)

```
SleepMessage( seconds, message [ , title [ , OK allowed?
            [ , condition ] ] ] )
```

Shows a wait message with a countdown.
If *OK allowed?* is 1, the dialog can be dismissed with a button, if not, this is not possible.
If a condition is given, it will be checked every second and the dialog will be closed as soon as it becomes fulfilled.

Example:

```
SleepMessage( 10, "Waiting for PocketWord", "Wait...", 0, \
              wndExists( "Word" ) )
```

9.9.3 Waiting for windows (WaitFor / WaitForActive)

See 9.6.7 Wait until a window exists (WaitFor) and 9.6.8 Wait until a window becomes active (WaitForActive)

9.10 Time

9.10.1 Unix timestamp (TimeStamp)

```
x = TimeStamp()
```

Returns the current time in seconds since 01/01/1970.

9.10.2 Formatted output

```
x = FormatTime( format [, timestamp ] )
```

Returns the time of the timestamp, or the current time if none is given, formatted corresponding to the format string.

These characters will be replaced with the corresponding value:

H	Hour (00-23)
h	Hour (01-12)
a	am/pm
A	AM/PM
i	Minute (00-59)
s	Seconds (00-59)
d	Day (01-31)
m	Month (01-12)
Y	Year (4 digits)
y	Year (2 digits)
w	Day of week (0=Sunday to 6=Saturday)
u	Unix timestamp

All other characters remain unchanged.

Examples:

```
x = FormatTime( "h:i:s a" )
x = FormatTime( "m/d/Y", TimeStamp() + 86400 )
```

9.10.3 Set current time to multiple variables (GetTime)

```
GetTime( variable, variable, variable )
```

Retrieves the current time into three variables for hour, minute, and seconds.

```
GetTime( variable, variable, variable,
         variable, variable, variable )
```

Like above, but three more variables for day (of month), month, and year (4 digits).

9.11 Copy, rename, move, and delete files

9.11.1 Copy a single file (Copy)

Copy(*source file*, *target file* [, *overwrite?*] **)**

Copies a file.
The target must contain a filename, too. (I.e., not only the path!)
If *overwrite?* is 0 or omitted, already existing files won't be overwritten.

Example:
```
Copy( "\My documents\test.txt", "\Storage\text.txt" )
```

9.11.2 Copy multiple files (XCopy)

XCopy(*source files*, *target directory* [, *overwrite?*] **)**

Copies files to the target directory.
The source can contain wildcards (* and ?) in the filename (e.g. "\My documents*.psw", but not "\My **.psw").
The target must be an existing directory.
If "overwrite?" is 0 or omitted, already existing files won't be overwritten.

Example:
```
XCopy( "\My documents\*.txt", "\Storage" )
```

9.11.3 Rename or move a single file (Rename)

Rename(*source file*, *target file* [, *overwrite?*] **)**

Renames or moves a file.
You have to include the path in the target, too!
If "overwrite?" is 0 or omitted, already existing files won't be overwritten.

9.11.4 Move multiple files (Move)

Move(*source files*, *target directory* [, *overwrite?*] **)**

Moves files to the target directory.
The source can contain wildcards (* and ?) in the filename (e.g. "\My documents*.psw", but not> "\My **.psw").
The target must be an existing directory.
If "overwrite?" is 0 or omitted, already existing files won't be overwritten.

9.11.5 Delete file(s) (Delete)

Delete(*files* **)**

Deletes the file(s).
The file parameter can contain wildcards (* and ?) in the filename (e.g. "\My documents*.psw", but not "\My **.psw").

9.11.6 Delete files, also in subdirectories (DelTree)

DelTree(*files* **)**

Deletes the file(s), including all subdirectories.
If the (sub)directory is empty afterwards, it will be removed.
The file parameter can contain wildcards (* and ?) in the filename (e.g. "\My documents*.psw", but not "\My **.psw"), which will also be used for the subdirectories.
Please handle with care!

9.11.7 Creating a shortcut/link (CreateShortcut)

CreateShortcut(*shortcut file, target file* **)**

Creates a shortcut (link) to the target file. This can be an entry in the start menu, for example.

Example:

```
CreateShortcut("\Windows\Startmenü\Test.lnk","\Storage\Test.exe")
```

9.12 Reading and writing text files

9.12.1 Reading a text file (ReadFile)

x = **ReadFile(** *file name* **)**

Reads the entire contents of the file into the variable. The file size is limited to 128kB or the available memory (whichever is less).
You can parse the file e.g. with ForEach line in split %contents%,"^LF^",1
See also the ForEach possibilities for INI files and ReadINI!

9.12.2 Writing to a text file (WriteFile)

WriteFile(*file name, contents* [, *append?*] **)**

Writes the contents to the file.
If *append?* is 0 or the parameter is omitted, an existing file will be overwritten, otherwise the given contents is appended at the end of the existing data.

9.12.3 Reading a value of an INI file (IniRead)

```
x = IniRead( file name, section, entry )
```

Reads an entry from an INI file. The section name must be passed without the brackets.

Example:

```
x = IniRead( "\My documents\test.ini", "Settings", "Test" )
```

9.12.4 Access serial ports (SetComInfo)

```
SetComInfo( port, timeout [, baud rate [, parity [, bits
            [, stop bits [, control ]]]]] )
```

With this command you define, how a COM port is accessed.
The command must be invoked before ReadFile or WriteFile. When you call those functions with „COM1:“ (or any other COM port) as file name, the access is initialized with the given values.

Parameters:

Port..... The port as DOS filename, e.g. "COM1:". Please regard uppercase and colons!
Timeout.....Timespan in milliseconds after which the system should cancel the access
Baud rate The transfer speed. Usually it's 9600, 14400, or 56700, the default if you omit this parameter is 9600.
Parity.....The parity of a check bit. Possible values are "none", "even", "odd", "mark", and "space". In most cases, it's "none", which is also the default, rarely "even" or "odd".
Bits.....Number of bits per transmitted byte. Nowadays it's almost always 8 (default), only in rare cases it's 7, less it almost never used.
Stop bits.....Number of stop bits (duration between bytes). Possible values are 1 (default) 1.5 (pass as quoted string!), and 2.
Control.....Type of flow control. Available are "None", "RTS/CTS" (default), and "XON/XOFF".

Hint: Depending on system, drivers, and device, not all parameters are always used correctly. Especially the timeout seems to be handled differently, sometimes it's even ignored completely.

9.13 File system informations

9.13.1 Check whether file or directory exists (FileExists/DirExists)

```
x = FileExists( file name )
x = DirExists( directory name )
```

Returns “1” (“true”), if the file or directory exists, “0” (“false”) if not.
It also returns “0”, if the entry doesn't correspond to the queried type. This means, “FileExists("\Windows")” is “false”, because it's a directory, not a file.

9.13.2 Check free space (FreeDiskSpace)

```
x = FreeDiskSpace( directory )
```

Returns the free disk space in the given directory in bytes (max. 2147483147 = ~2GB).
On Windows Mobile devices, the directory (e.g. “\Storage“ for the storage card), on PCs the drive letter (“D:\...”) is regarded.

9.13.3 Get file size (FileSize)

```
x = FileSize( file name )
```

Returns the size of the file in bytes.

9.13.4 Get file creation time (FileCreateTime)

```
x = FileCreateTime( file name )
```

Returns the creation time of the file as unix timestamp, or 0 if the file doesn't exist.
See also 9.10 Time for informations about how to compare or format it.

9.13.5 Get file modification time (FileModifyTime)

```
x = FileModifyTime( file name )
```

Returns the last modification time of the file as unix timestamp, or 0 if the file doesn't exist.
See also 9.10 Time for informations about how to compare or format it.

9.13.6 Get file attributes (FileAttribs)

`x = FileAttribute( file name, attribute )`

Returns the current state of the given file attribute. 1=attribute is set, 0=not set

Allowed values for "a*ttribute"*:

- `directory` (is the given file a directory?)
- `hidden` (hidden file?)
- `readonly` (write protection?)
- `system` (system file?)
- `archive` (not archived?)

Must be passed as string (i.e. in quotation marks or as value of a variable/expression).

9.13.7 Set file attributes (SetFileAttribute, SetFileAttribs)

`SetFileAttribute( file name, attribute, set? )`

Sets (*set?*=1) resp. removes (*set?*=0) the given file attribute. All other attributes remain unmodified.

Allowed values for "a*ttribute"*:

- `hidden` (hidden file?)
- `readonly` (write protection?)
- `system` (system file?)
- `archive` (not archived?)

Must be passed as string (i.e. in quotation marks or as value of a variable/expression).

Examples:

```
SetFileAttribs("\Test.txt", "hidden", 1)
```

→ hides the file

```
SetFileAttribs("\Test.txt", "readonly", 0)
```

→ removes the write protection

`SetFileAttribs( file name, read only? [, hidden? [, archive? ]] )`

Sets the given file attributes. With 1 (or any other numeric value except 0) the attribute is set, with 0 it'll be removed. An empty string ("") will keep the attribute unmodified.

Examples:

```
SetFileAttribs("\Test.txt", "", 1)
```

→ hides the file, read only (and other attributes) remains unmodified

```
SetFileAttribs("\Test.txt", 0)
```

→ removes read only attribute, all other attributes remain unmodified

9.13.8 Get version number (FileVersion / GetVersion)

```
x = FileVersion( file name )
GetVersion( file name, variable, variable, variable, variable )
```

Gets the version number in the resources, either as string ("a.b.c.d") or into single variables. This information isn't contained or accurate in all files. If contained, the version is always in four levels, usually major, minor, patch, and build version.

When using the function FileVersion, the parts are concatenated with dots, (e.g. "3.1.2.0"), the command GetVersion assigns each part to a separate variable.

9.14 ZIP archives

9.14.1 Important hints

With the functions, which are currently available to me, it is not possible to overwrite files contained in an archive. If an already existing file is added again, it's really added another time to the archive, i.e., there are two entries for the same file. Not all packers cope with something like that. Due to this, I recommend to create a new archive if you're in doubt..

Another problem is the encoding of file names in ZIP archives. There is no standard for that, and unicode is not supported.
MortScript uses – like most Windows/DOS programs – the DOS code page 437. This might cause troubles if your files contain special characters or foreign languages (e.g. Cyrillic or Greek characters).

9.14.2 Compress a single file (ZipFile)

ZipFile(*source file*, *ZIP file*, *file name in archive* [, *rate*] **)**

Adds the given file to the archive. The *source file* (the file to compress) and the *ZIP file* must be given with the complete path. In spite to this, the file name in the archive is usually with a relative path, or completely without a path.

The compression rate ranges from 1=no compression to 9=best, if omitted, it defaults to 8.

Example:

```
ZipFile( "\Storage\Test\manual.psw", "\Storage\mans.zip", \
         "test\testman.psw" )
```

9.14.3 Compress multiple files (ZipFiles)

```
ZipFiles( source files, ZIP file [, subdirectories?
          [ , path in archive [, rate] ] ] )
```

Adds the given files to the archive. The *source files* are given, like for XCopy or Move, with a fixed path and wildcards in the file name (e.g. "\My documents*.psw").
If *subdirectories?* is "1", the filename filter is also used for subdirectories, i.e., "\My documents*.psw" would include "\My documents\Word\x.psw", too.
In the archive, the given path of the source files is omitted, and – if one is passed – prefixed with the *path in archive*. I.e., if *no path in archive* is given, "\My documents\Word\x.psw" will become "Word\x.psw" in the archive, "\My documents\x.psw" will become "x.psw", etc. If the *path in archive* was e.g. "docs", the file names in the archive would become "docs\Word\x.psw" resp. "docs\x.psw".

Examples:

```
ZipFiles( "\Storage\Test\*.psw", "\Storage\mans.zip", 1, "test" )
```

→ Compresses all *.psw files from \Storage\Test and subdirectories to the directory "test" in the archive \Storage\mans.zip

```
ZipFiles( "\Storage\Test\*.jpg", "\Storage\jpgs.zip" )
```

→ Compresses all *.jpg files from \Storage\Test to the main directory of the archive \Storage\jpgs.zip. Subdirectories are ignored.

9.14.4 Extract single file (UnzipFile)

```
UnzipFile( ZIP file, file name in archive, target file )
```

Unzips the given file.
The target file must be given with complete path, the path of the compressed file will be ignored for the target file.

Example:

```
UnzipFile( "\Storage\mans.zip", "test\test.psw", \
           "\Storage\test.psw" )
```

→ Unzips the file „test\test.psw" of the archive "\Storage\mans.zip" to "\Storage\test.psw"

9.14.5 Extract entire archive (UnzipAll)

```
UnzipAll( ZIP file, target directory )
```

Unzip all files contained in the archive to the given directory. Paths contained in the archive will be used and, if necessary, created.

9.14.6 Extract a path of an archive (UnzipPath)

UnzipPath(*ZIP file*, *path in archive*, *target directory* **)**

Unzips all files located in the given path in the given archive.
Subdirectories of that path are unzipped, too. The given path name is not created in the target directory, but its subdirectories will. The target directory must exist.

Example:
```
UnzipPath( "\Storage\mans.zip", "test", "\Storage\test-unzip" )
```
→ Unzips all files contained in the directory “test” and its subdirectories of the archive “\Storage\mans.zip” to “\Storage\test-unzip”. I.e., “test\sub\x.psw” would be extracted to “\Storage\test-unzip\sub\x.psw”.

9.15 Connections

9.15.1 Establish connection (Connect)

```
Connect
Connect( connection name )
Connect( title, message )
```

Connects to the Internet.

Connect without a parameter tries to use the default connection, but at least on some PPCs this doesn't work reliable.

Connect with a connection name uses the given connection. The connection names can be set freely in the system's settings and are usually initialized by your operator. The Internet connection is usually named “Internet”, “The Internet” or similar.

If a title and message are given as parameters, all available connections are listed (like using Choice), and the chosen one is used.

Not available for: PC, PNA

9.15.2 End connection (CloseConnection/Disconnect)

```
CloseConnection
Disconnect
```

CloseConnection releases a connection that's been established with Connect. This only signals the system, MortScript won't use this connection anymore. It's up to the system how it reacts to this, so the connection might stay established.

In spite to this, Disconnect terminates all connections, including ActiveSync. Sadly, since WM5 AKU3, this doesn't work anymore. Currently, there's no known way to hang up a connection from a program.

Not available for: PC, PNA

9.15.3 Check connection (Connected/InternetConnected)

```
x = Connected()
x = InternetConnected( [ URL ] )
```

Connected checks, whether there is an “RAS connection” (“Remote Access”). This is always the case for ActiveSync connections, for other connections on most devices – but not all...
Returns „1“ if a connection exists, „0“ if not.

InternetConnected checks, whether a connection to the Internet exists. Sadly, most devices return “true” for a pure connection check (i.e., the function returns “1”) and checks the connection only if a target server is accessed. Due to this, you can pass an URL, which will be used for a connection test (e.g. "http://www.google.com").

Not available for: PC, PNA

9.16 Internet access

9.16.1 Set proxy

```
SetProxy( proxy )
```

Set the proxy for http access. Using Windows Mobile, it's not quite easy to use the proxy from the system configuration...
Should be something like "proxy.foo.bar:8080".

Not available for: PNA

9.16.2 Download (Download)

```
Download( URL, target file )
```

Similar to Copy, but uses an URL ("http://..." or "ftp://...") as source and shows a progress window, because this usually takes a bit longer...

Example:

```
Download( "http://www.sto-helit.de/test.txt", \
          "\Storage\text.txt" )
```

Not available for: Smartphone, PNA

9.16.3 Other possibilities

All file operations reading a single file, will also work with an URL as source file. This regards Copy, ReadFile, IniRead, and some ForEach variations.

9.17 Directories

9.17.1 Create directory (MkDir)

MkDir(*directory* **)**

Creates the directory.
It's not possible to create multiple levels at once!
I.e., MkDir("\My documents\Some\Path") will fail if the "Some" subdirectory does not already exist.

9.17.2 Remove directory (RmDir)

RmDir(*directory* **)**

Removes the directory.
There mustn't be any files or subdirectories contained in the folder.

9.17.3 Change directory (ChDir)

ChDir(*directory* **)**

Makes the directory the current directory.
Only available for PC version, Windows Mobile hasn't a “current directory” concept.

9.17.4 Getting system paths (SystemPath)

x = **SystemPath(** *type* **)**

Receives the localized directory name for certain locations.
The type must be given as string value, e.g. "StartMenu".

Possible values:
ProgramsMenu “Programs” in the start menu
StartMenu..... The start menu, doesn't work on Smartphones
Startup..... Startup folder (entries are run after soft reset resp. if Windows is started)
Documents.....”\My documents” or localization, doesn't work on devices with PPC2002
ProgramFiles..... ”\Program files” or localization, doesn't work on devices with PPC2002
ScriptExe..... Path to MortScript.exe (without file name), doesn't work on Smartphones
ScriptPath..... Path to the current script (without file name)
ScriptName..... Name of the current script (no path+extension)
ScriptExt..... Extension of the current script (".mscr" or ".mortrun").

I.e., you could execute the current script with

```
Run ( SystemPath("ScriptExe") \ "MortScript.exe", \
      SystemPath("ScriptPath") \ SystemPath("ScriptName") & \
        SystemPath("ScriptExt") )
```

9.18 Registry

9.18.1 Reading registry entries (RegRead)

```
x = RegRead( root, key, value name )
```

Reads the given value from the registry.

For *root* these values are allowed:
HKCU..... HKEY_CURRENT_USER
HKLM..... HKEY_LOCAL_MACHINE
HKCR..... HKEY_CLASSES_ROOT
HKUS..... HKEY_USERS
Only the four letter abbreviations are supported! Remember to use quotes, if you don't want to use a probably undefined variable like HKCU.

If the *value name* is an empty string (""), the default value is used. (In registry editors usually displayed as ”<Default>” or ”@”).

The value's data type is automatically regarded. DWords are returned as number, string values as strings, and binary data as a string containing the data as hex dump (e.g. "010ACF").

9.18.2 Writing registry entries (RegWriteString/RegWriteDWord/ RegWriteBinary)

```
RegWriteString( root, key, value name, value )
RegWriteDWord( root, key, value name, value )
RegWriteBinary( root, key, value name, value )
```

Writes a value to the registry.

Valid values for *root* are listed at 9.18.1 Reading registry entries (RegRead).
If the *value name* is an empty string (""), the default value is used.

RegWriteString writes a string value (a numeric value is automatically converted).
RegWriteDWord writes a numeric value (a string value is automatically converted, invalid strings will become “0”).
RegWriteBinary writes binary data. The given value must be a string containing a hex dump (e.g. "010A"), spaces and similar characters are not allowed in it!

Examples:

```
RegWriteDWord( "HKCU", "Software\Microsoft\Inbox\Settings", \
               "SMSDeliveryNotify",1 )
```

(Delivery notification for SMS on many phone edition devices)

```
RegWriteString( "HKCU", "Software\Mort\MortPlayer\Skins", \
                "Skin", "Night" )
RegWriteBinary( "HKCU", "Software\Mort\Dummy", "", "C000" )
```

9.18.3 Checking existence of a value (RegValueExists)

```
x = RegValueExists( root, key, value name )
```

Returns 1, if the given value exists, 0 if it doesn't.
Values for *root* are as in RegRead.

9.18.4 Checking existence of a key (registry path) (RegKeyExists)

```
x = RegKeyExists( root, key )
```

Returns 1, if the given key (a “subdirectory” in the registry) exists, 0 if it doesn't.
Values for *root* are as in RegRead.

9.18.5 Removing a registry value (RegDelete)

```
RegDelete( root, key, value name )
```

Removes the registry value.
Values for *root* are as in RegRead.

9.18.6 Removing a registry key (registry path) (RegDeleteKey)

```
RegDeleteKey( root, key, values?, sub keys? )
```

Removes an entire key (a “subdirectory” in the registry).
If *values?* is 1, all contained values are removed, too.
values? is also used for sub keys if *sub keys?* is 1.
I.e., RegDeleteKey("HKCU", "\Software\Something", NO, YES) would remove only empty sub keys (because sub keys with entries can't be deleted).

If the key can't be removed, a message is shown if the ErrorLevel's on "warn" or lower.

The path mustn't be empty to avoid accidental removing of an entire registry section (e.g. if there's a typo in a variable). Still, this command has to be handled with care, esp. when using variables or expressions for the path!

9.19 Dialogs

9.19.1 Free text input (Input)

```
x = Input( message [, title [, numeric? [, multiline?
          [, default ]]]] )
```

Opens a simple dialog that allows to enter any text, which will be returned by the function.
If *numeric?* is „1“, only digits can be entered (no ”-” or ”.”, too!).
If *multiline?* is „1“, a multi line text box is displayed. On most devices, *numeric?* will be ignored if this option is used.
If you've given a *default*, it will be shown in the edit box.

9.19.2 Message (Message)

```
Message( text [, title ] )
```

Shows the given text in a message window.

9.19.3 Big message with scrollbar (BigMessage)

```
BigMessage( text [ , title ] )
```

Similar to Message, but doesn't use the system's message box function, which resizes to the contained text (except for Smartphones). Instead, a fixed sized internal dialog is used, which shows the text in a scrollable text box.

9.19.4 Message with countdown/condition (SleepMessage)

```
SleepMessage( seconds, message [ , title [ , OK allowed?
            [ , condition ] ] ] )
```

See 9.9.2 Wait message with countdown / condition (SleepMessage)

9.19.5 Simple questions (Question)

```
x = Question( question [, title [, type ] ] )
```

Shows a simple question. This uses a default dialog from the system, so the button labels are localized by Windows.

Allowed types:
"YesNo".....Shows “Yes” and “No” (default)
"YesNoCancel".....Shows “Yes”, “No”, and “Cancel”
"OkCancel".....Shows “OK” and “Cancel”
"RetryCancel".....Shows “Retry” and “Cancel”

The type must be a string, so you have to use quotes ("YesNo") or e.g. a variable assigned with the wanted type string.

Rückgabewerte:
“Yes”, “OK”, “Retry”: 1 (predefined variable “YES”)
“No”: 0 (predefined variable “NO”)
“Cancel”: 2 (predefined variable “CANCEL”)

Be aware, that “Cancel” would be a fulfilled condition in statements like If or While, so you either have to use something like “If (Question(...., "OkCancel") = CANCEL)” or “Switch(Question(....))” to handle it correctly.

9.19.6 Selection from a list (Choice)

```
x = Choice( title, hint, Standard, timeout, value, value
            {, value} )
x = Choice( title, hint, Standard, timeout, array )
```

Works similar to 8.4 ChoiceDefault, but returns the selected entry instead of starting a control structure.
I.e., Switch(Choice(...)) and ChoiceDefault(...) do the same.

It's handy to use Choice as function, if the value is required later on or at different locations.

9.20 Processes (running applications)

9.20.1 Checking existence of a process (ProcExists)

```
x = ProcExists( process name )
```

Returns 1, if the given process is running, 0 if not.

The “process name” is the name of the EXE without path, e.g. "solitaire.exe".

9.20.2 Checking existence of a script process (ScriptProcExists)

```
x = ScriptProcExists( script name )
```

Returns 1, if the given MortScript is running, 0 if not.

“ProcExists” isn't working for MortScripts, because the process name is always “MortScript.exe”.

The script name can be either the script name without path (e.g. “myscript.mscr”) or with the full path (e.g. “\My documents\myscript.mscr”), in the PC version, also the drive letter is required.

See also informations in 9.20.5 End a running script (KillScript).

9.20.3 Process name of active window (ActiveProcess)

```
x = ActiveProcess()
```

Returns the program name of the currently active window.

9.20.4 End a running process (Kill)

```
Kill( process name )
```

Terminates the application. The parameter must be the name of the exe without path (e.g. solitare.exe).

WARNING: This command kills the process regardless of any losses!

It could cause data loss, crashes, or error messages.

Wherever possible, you should use Close instead, which allows the application to end gracefully (save/close files, etc.).

9.20.5 End a running script (KillScript)

KillScript(*script name* **)**

Ends the given script. KillScript waits up to 3 seconds for the current command of the script to be finished, to avoid troubles with improperly terminated actions. If that doesn't work, the process is terminated similar to Kill.

The script name can be either the script name without path (e.g. “myscript.mscr”) or with the full path (e.g. “\My documents\myscript.mscr”), in the PC version, also the drive letter is required.

If the script name is given without path, it's possible that another script with the same name as the one you wanted, but from another path, is running. So it might be a good idea to give the full path if it's possible (e.g. with SystemPath).

Keep in mind a script can't be run twice. If you want to start and stop a background task, it might be a good idea to create an additional script that starts the task (Run command) if it isn't running (use ScriptProcExists for that), and kill the script with KillScript if it is running.

Do NOT use RunWait or CallScript, because with this, the invoking script would remain active until the background task is finished, and thus couldn't be invoked a second time to kill the background script!

Example:

```
backScript = SystemPath( "ScriptPath" ) \ "background.mscr"
If ( ScriptProcExists( backScript ) )
    If ( Question( "Stop background process?" ) = YES )
        KillScript( "background.mscr" )
    EndIf
Else
    Run( backScript )
EndIf
```

9.21 Signals

9.21.1 Modifying the system volume (SetVolume)

SetVolume(*value* **)**

Sets the system volume to the given value. Possible values are 0 (off) to 255 (loudest).
Some devices, like the Loox720, round to certain steps between (usually 4 or 16 levels), most devices really support all 256 levels.

Not available for: PC

9.21.2 Play a WAV file (PlaySound)

PlaySound(*WAV file* **)**

Plays the given file. The script is paused until the sound is played.

9.21.3 Vibrate (Vibrate)

Vibrate(*milliseconds*)

Lets the device vibrate for the given time.
The PC version beeps instead.
For PPCs, the vibrator is accessed like an status LED, but there's no standard for its number. MortScript assumes it's the last available “LED”, which seems to work for most devices. But it might also happen, that some LED lights up or nothing happens at all.

9.22 Display / screen

9.22.1 Get the color at a screen position (ColorAt)

x = **ColorAt(** *x*, *y* **)**

Gets the color of the screen's pixel at the given position. At least on some devices, the title bar is ignored, i.e. it receives the color of the underlying today background.

9.22.2 Create the color code from RGB values (RGB)

x = **RGB(** *red*, *green*, *blue* **)**

Converts the decimal values of red, green, and blue parts (0-255 each) into the the same format as used by ColorAt(...), so it's useful for comparisons.

9.22.3 Rotate the screen (Rotate)

Rotate(*orientation* **)**

Rotates the screen.
Valid values are: 0=default (portrait), 90=right handed, 180=upside down, 270=left handed

Not available for: Smartphone, PC, PPC/PNA with WM2003 or below

9.22.4 Set backlight intensity (SetBacklight)

SetBacklight(*battery, external* **)**

Sets the brightness of the backlight to the given values.
Battery is for battery power, *external* for external power.
Valid values are between 0 and 100.
This command will not work on all devices!
Also, the value for the highest luminance differs for each device. Currently, I know about devices with 10, 60, or 100 as highest possible value, some devices even work the other way around (e.g. 10=off, 0=brightest) or have an exception for the brightest value (e.g. 0=brightest, 1=darkest, 10=next to brightest).

Not available for: PC, PNA, Smartphone

9.22.5 Toggle display on/off (ToggleDisplay)

ToggleDisplay(*on?* **)**

Turns the display on (*on?* = 1) or off (*on?* = 0).

Not available for: PC, PNA, Smartphone

9.22.6 Check screen informations (orientation/resolution) (Screen)

x = **Screen(** *type* **)**

Returns 1, if the screen fulfills the *type* condition, 0 if it doesn't.
Allowed values for *type*:
"landscape" (in landscape orientation?)
"portrait" (in portrait orientation?)
"vga" (VGA resolution – no matter if “default” or “real/true VGA”)
"qvga" (QVGA resolution)

Not available for: PC

9.22.7 Heute-Bildschirm aktualisieren (RedrawToday)

RedrawToday

Redraws the Today screen. Useful if you did any modifications in the registry...
Not available for: PC

9.22.8 Show/hide wait cursor (ShowWaitCursor/HideWaitCursor)

```
ShowWaitCursor
HideWaitCursor
```

Shows (ShowWaitCursor) or hides (HideWaitCursor) the hourglass (or rotating disc in Windows Mobile).

9.23 Clipboard

9.23.1 Copy text to the clipboard (SetClipText)

```
SetClipText( text )
```

Copies the given text to the clipboard.

9.23.2 Get text from the clipboard (ClipText)

```
x = ClipText()
```

Returns the text from the clipboard.
It is a precondition, that a text variant of the copied data is available in the clipboard. This depends from the application that filled the clipboard.

9.24 Memory

9.24.1 Available main memory (FreeMemory)

```
x = FreeMemory()
```

Returns the available main memory in kilobytes.
Devices with an older system than Windows Mobile 5 dynamically split the device's memory between memory for programs (FreeMemory()) and a “RAM disk” for the main directory (FreeDiskSpace("\")).

9.24.2 Size of the main memory (TotalMemory)

```
x = TotalMemory()
```

Returns the total size of the main memory in kilobytes.

9.25 Energy

9.25.1 Check if externally powered (ExternalPowered)

x = **ExternalPowered()**

Returns 1, if the device is externally powered, 0 if a battery is used.

The PC variant always returns 1, even if it's a notebook powered by battery.

9.25.2 Current battery level (BatteryPercentage)

x = **BatteryPercentage()**

Returns the current battery level in percent. If the device is externally powered, this value might be incorrect on some devices.

The PC variant always returns 100, even if it's a notebook powered by battery.

9.25.3 Turn off device (PowerOff)

PowerOff

Turns off the device (to standby mode). After power on, the script is continued. Be aware that accessing the storage cards usually doesn't work directly after power on. Thus, a Sleep and/or While(not FileExists(...)) might be useful to avoid errors...

Not available for: PC

9.25.4 Avoid automatic power off (IdleTimerReset)

IdleTimerReset

Resets the idle timer of the system. This way, you can avoid (if called in a loop) or postpone the automatic power off.
Sadly, the system uses another timer for turning off the background light, which can't be queried or modified (at least there's no documentation about it). So there's no way to avoid that.
Also, there seem to be some rare devices where IdleTimerReset has no effect.

Not available for: PC

9.26 System

9.26.1 Get the system version (SystemVersion)

```
x = SystemVersion( [element] )
```

Returns the version of the system. If no or an invalid parameter is given, it returns a string in the format major.minor.build, for example 5.1.2600 for XP with SP2 or 5.1.195 for WM5.

With a parameter, you can get single parts. Available parameters:
"major".....returns the major version number
"minor"..... returns the minor version number
"build".....returns the build number
"platform"..... returns the platform, one of "Win95" (includes 98 and Me), "WinNT" (includes XP and Vista), and "WinCE" (includes Smartphone / PPC / Windows Mobile).

9.26.2 Get the current MortScript variant (MortScriptType)

```
x = MortScriptType()
```

Returns, which MortScript variation is used to execute the script. Currently, there are these return values:
"PPC"..... PocketPC
"PC"..... PC (Desktop)
"SP".....Smartphone
"PNA"..... PocketNavigation (downsized Windows Mobile devices for navigation)

9.26.3 Restart device (Reset)

```
Reset
```

Executes a soft reset. Please use it only in scripts used only by yourself or after a warning / query.

Not available for: PC

10 Old syntax and commands

The following syntax, conditions, and commands are still supported for compatibility reasons, **but shouldn't be used any longer.**

They're primarily listed in this manual to allow you to understand (and maybe convert) old scripts.

10.1 Old syntax

In older versions, the syntax was different:

Variables always had to be enclosed in %...%, except when they were assigned by a command (like Set or GetTime).

Command parameters weren't given in parentheses, and the parameters weren't expressions by default but divided as follows:

- { ... }: An expression
- %...%: A variable
- "...": A fixed string
- everything else: An unquoted string, that ends et the next comma or end of line. Surrounding spaces/tabulators are removed. For commands assigning a variable (like GetTime) this might also be variable names. %...% caused the variable's contents to be used as variable name.

Example:

```
Copy \Storage\test.txt, { %zielpfad% \ "test.txt" }, %overwrite%
```

instead of

```
Copy( "\Storage\test.txt", zielpfad \ "test.txt", overwrite )
```

10.2 Old conditions

The old condition syntax is alternative to parentheses, i.e., something like „If wndExists "Window"", **not** „If (wndExists "Window")".
The parameters for those old conditions aren't expressions, and mustn't be given in parentheses! (Except for { ... } and "expression")

expression *expression*
{ *expression* **}**

Alternative styles for the now common **(** ... **)**
{...} was done to allow a style similar to the old expression syntax for parameters.
Checks the given expression.

equals *value1, value2*

Returns true if the two values are equal. Usually only makes sense, if at least one of the parameters is a variable.

fileExists *file*

Checks whether the given file exists. If the parameter identifies a directory, the condition will be false!

dirExists *directory*

Checks whether the given directory exists. If the parameter identifies a file, the condition will be false!

procExists *application*

Checks whether the given application is running. The parameter must be the name of the exe without path (e.g. solitare.exe).

wndExists *window title*

Checks whether a window exists, which includes with the given string in it's title. E.g. “If wndExists Word” will be true if a window named "PocketWord" exists.

wndActive *window title*

Similar to “wndExists”, but it's only true if the given window is the active (foreground) window.

question *text*[, *title*]

Will show a simple Yes/No dialog with the given text (Yes/No will be localized by Windows). The condition's true if "Yes" was chosen.

screen landscape|**portrait**|**vga**|**qvga**

Checks whether the display is in the given mode.
Be aware "screen vga" will be true if a VGA display is used, no matter if "double resolution" (WM2003 SE default) or "real VGA" (SE_VGA, OzVGA, ...) is used.

regKeyExists *root*, *key*, *value name*

Is true if the given registry key exists. Parameters are like those of RegDelete.

regKeyEqualsDWord *root*, *key*, *value name*, *value*
regKeyEqualsString *root*, *key*, *value name*, *value*

Is true if the given registry key contains the given value.

Each condition can be negated with the prefix “not”.
E.g. “If not screen landscape” will be the same as “If screen portrait”, and “If not fileExists \Windows\some.dll” will be true if the given file doesn't exist.

10.3 Old commands

Input(*variable*, *numeric?*, *message* [, *title*] **)**
→ Like "Input", result is saved in given variable instead of being returned

SubStr(*string*, *Start*, *length*, *variable* **)**
→ Like "SubStr", result is saved in given variable instead of being returned

GetPart(*string*, *separator*, *trim?*, *index*, *variable* **)**
→ Like "Part", result is saved in given variable instead of being returned

Find(*string*, *search string*, *variable* **)**
→ Like "Find", result is saved in given variable instead of being returned

ReverseFind(*string*, *character*, *variable* **)**
→ Like "ReverseFind", result is saved in given variable instead of being returned

GetRGB(*red*, *green*, *blue*, *variable* **)**
→ Like "RGB", result is saved in given variable instead of being returned

MakeUpper(*variable* **)**
MakeLower(*variable* **)**
→ Similar to "ToLower" resp. "ToUpper", modifies the contents of the given variable

Eval(*variable*, *expression as string* **)**
→ Like "Eval", result is saved in given variable instead of being returned

GetColorAt(*x*, *y*, *variable* **)**
→ Like "ColorAt", result is saved in given variable instead of being returned

GetWindowText(*x*, *y*, *variable* **)**
→ Like "WindowText", result is saved in given variable instead of being returned

GetClipText(*variable* **)**
→ Like "ClipText", result is saved in given variable instead of being returned

GetActiveProcess(*variable* **)**
→ Like "ActiveProcess", result is saved in given variable instead of being returned

GetTime(*variable* **)**
→ Like "TimeStamp", result is saved in given variable instead of being returned

GetTime(*format*, *variable* **)**
→ Like "FormatTime" without timestamp, result is saved in given variable instead of being returned

GetActiveWindow(*variable* **)**
→ Like "ActiveWindow", result is saved in given variable instead of being returned

IniRead(*file*, *section*, *entry*, *variable* **)**
→ Like “IniRead”, result is saved in given variable instead of being returned

ReadFile(*file*, *variable* **)**
→ Like “ReadFile”, result is saved in given variable instead of being returned

GetSystemPath(*path type*, *variable* **)**
→ Like “SystemPath”, result is saved in given variable instead of being returned

GetMortScriptType(*variable* **)**
→ Like “MortScriptType”, result is saved in given variable instead of being returned

RegReadString(*root*, *key*, *value name*, *variable* **)**
RegReadDWord(*root*, *key*, *value name*, *variable* **)**
RegReadBinary(*root*, *key*, *value name*, *variable* **)**
→ Similar to “RegRead”, data type in the registry must be given, result is saved in given variable instead of being returned

11 Donations

Well, of course this program is freeware, so you don't need to pay anything.
But if you think “Wow, what a great program, I like to give him some money for his efforts”, I thankfully accept that.

Just go to www.paypal.com, register or log on, and send the money to “mort@sto-helit.de”.

You can get my bank account data by request, I don't like to publish it for everyone to see... And it's only useful if you're living in the EU.