MortScript V4.3

(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 – also see 3 Installation)

The script language is 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.

You have to use downloaded .mscr or .mortrun script files, or write them yourself with any text editor (see 4.1 Create and execute scripts). 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 grant 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 freeware, i.e., you don't need to pay for it, but I'd be glad about a small (or big ;)) donation as "thank you!". See also 11 Donations.

The source is available on request (SubVersion repository), but unlike e.g. GPL, you're not allowed to create your own derivate – that's to avoid a “For that script, you've got to use the XxxScript interpreter from Yyy” confusion.

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“).

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 For-, ForEach, While or Repeat loops
- Some system features (e.g. rotation, volume, backlight brightness, soft reset)
- Sub functions, local variables, multi level arrays, ...

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 9.27.2 Get the current MortScript variant (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-4.1-*system*.exe (e.g. MortScript-4.1-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

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

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).

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 is 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". Since V4.1, it's also possible to use Unicode files with proper prefixes (see 9.13.1 Reading a text file (ReadFile)) but ANSI is still recommended for backward compatiblity.

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.

All other parameters not starting with "/" or "-" are assigned to argv[x] in order of appearance and counted in argc.

The variables assigned by name=value are global values, so if you use local variables, you might need to exclude them with Global(...). Also see 7.5.2 Variable scope.

Opposed to that, argv and argc are only available in the main script body (outside of any Sub) to avoid complications with function parameters (see 8.11 Sub routines (Sub, Call/CallFunction, @...)).

Example:

```
path\to\MortScript.exe "path\to\script.mscr" opt1 message="Test" opt2
```

Will define the global variable "message" with value "Test", the array "argv" with element 1 being "opt1" and element 2 being "opt2", and the parameter counter "argc" with 2.

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 9.21.7 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 tools 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 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".

Line breaks in a string must be entered as ^NL^, see 7.3 Fixed strings.

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 should always be given with complete path (e.g. `"\path\to\file.ext"` or `"\some\directory"`), because Windows Mobile doesn't know a "current directory" - and on the desktop it could not be what you expected.

If no path is given, MortScript adds the directory of the currently running script – but only when MortScript accesses the file contents, not for system operations. I.e., ReadFile, WriteFile, ReadIni, WriteIni, ForEach with IniSections or IniKeys, and CallScript(Function) will work with relative paths, but not commands like Copy, Move, Rename, all ZIP operations, and so on.

Or in short: Everything that's good to configure your script can be read from and written to the script's directory, everything else needs full paths.

Windows Mobile doesn't support "." and ".." in paths, not even something like "\some\path\..\file.txt" (which might be the result of adding the script's path, or if you use SystemPath and add something), i.e. you can't access files from a parent directory that way.

6.6 Comments

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

In INI files, comments are possible with a ";" at the beginning of the line (as usual with INI files...).

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 “BatteryPercentage()” 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 support “typing”, i.e. you can't tell variable “x” only to contain numbers while “str” only contains strings. Numbers and strings are stored differently, but are automatically converted to each other when necessary. There are only some rare conditions where the current value's type is considered, like when using comparisons.

Whether a value is interpreted as number or string usually 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”.

If a string contains a decimal point, it's converted to a float value, and, if an integer is expected, rounded. E.g. “SleepMessage("4.5", "Waiting...", "Wait")” will wait for 5 seconds. If a floating point value is converted to a string (e.g. for Message), it's formatted to contain 6 decimals by default. You should use the Format() function instead of the default conversion to get better results.

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 = `**`5.4321`**, `Sleep(`**`20`**`)`, ...).
If you want floating point operations, you must use a decimal point, like “1.” instead of “1”.

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.

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

7.5.1 Predefined variables

Some variables are predefined, to allow better readable instructions. Contrary 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.

`PI` is initialized with 3.14159265358979323846264338327950 (π)

`SQRT2` is initialized with 1.41421356237309504880168872420970 (square root of 2)

`PHI` is initialized with 1.61803398874989484820458683436560 (φ)

`EULER` is initialized with 2.71828182845904523536028747135270 (*e*)

7.5.2 Variable scope

Usually, all variables are global, this means, if you set a variable, it's value can be queried and modified in all Sub blocks (see 8.8 Sub routines (Sub / Call)) and scripts invoked by CallScript (see 8.9 Other script as subroutine (CallScript)).

If you want to use local variables, you've got to use either the command “Local” or “Global”.

If you invoke Local() without parameters, all used variables are treaded as local starting with that command, until the Sub block or script ends. This goes for the main routine (the code before the first Sub), too.

You can also pass a list of variables to Local(), so only those variables will be local while everything else remains global.

Global() works the other way around: The list of variables passed to this command will remain global, while everything else will be local. It's like Local() with an exception list.

Example:

```
Local()
x = "Test"
Call( "Sub1" )
Call( "Sub2" )
Message( x )       will show „Test“, because the local variable can't be changed in subs
Message( y )       will show nothing, because there's no local variable y (only global, see below)

Sub Sub1
  x = 5            will set the global variable
  Local( x )
  y = "Hi!"        global variable! (only x is local)
  Message( x )     will show nothing (the local variable's not initialized!)
EndSub

Sub Sub2
  Global( x )
  Message( x )     will show the global value „5“
  Message( y )     will show nothing, because there's no local variable y
EndSub
```

7.5.3 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.

Numbers given as string are converted to numerical indexes (and back) unless they start with 0. This is done so it's possible to separate arr["007"] from arr["7"]. Opposed to that, arr[7], arr[007], and arr["7"] do address the same element. You might need to use tricks like arr[input+0] to enforce numerical indexes.

You can use additional levels by adding additional brackets, for example “colors[x][y]”.

Examples:

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

list[1] = "a"
list[2] = "b"
list["3"] = "c"
list["0004"] = "d"
list[5] = "f"
list["a"] = "A"
idx = Choice( "Selection", "Choose something", 0, 0, list )
```

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

7.5.4 References ([*variable name*])

References allow you to access a variable (or array element) by an expression.

The can be viewed like some kind of mixture between seeing the variables as an unnamed array and Eval() (see 9.2.2 Expressions in a string (Eval)).

To refer to a variable, just use an expression that evaluates to a valid variable name (optionally with array element) in brackets.

For example, “["array[1]"]” will refer to the first element of “array”. Of course, this doesn't make much sense in that way, because “array[1]” would be the same and faster to parse. But make it e.g. “[arrayName & "[" & elem & "]"]”, and you'll see the advantage.

You can use this almost everywhere, except for commands in the old syntax without parentheses (see 10 Old syntax and commands).

<u>Examples:</u>

```
[targetVar] = [sourceVar] * 10
Choice( "Selection", "Select:", [choiceArrayName] )
GetTime( [varHour], [varMinute], [varSecond] )
```

7.6 Operators

7.6.1 List of all possible operators

<u>All possible operators by priority (highest first):</u>

`()`	Parentheses
`NOT`	Negation
`^`	Power (x^y → x^y)
`* / MOD`	Multiplication, division, modulo (remainder of divisions)
`+ -`	Addition, subtraction
`& \`	Concatenation of strings
`> >= < <= = <>`	Numerical comparisons
`gt ge lt le eq ne`	Alphanumerical comparisons
condition ? *true* : *false*	Returns the “true” value if the condition is fulfilled, otherwise the “false” value
`AND &&`	Binary / logical “and”
`OR ||`	Binary / logical “or”

7.6.2 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”.

7.6.3 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.

7.6.4 Concatenation of strings and paths

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

<u>Example:</u>
```
"\My documents\" \ "\file.txt"
"\My documents" \ "file.txt"
"\My documents\" \ "file.txt"
```
→ will all result in `"\My documents\file.txt"`.

Compared 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, also predefined as variables FALSE and NO).
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 }
{ ElseIf( 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 ElseIf is used, only the lines between the **first** true condition (including the initial If) and the upcoming ElseIf, Else resp. EndIf are executed. Only if **no** condition is fulfilled, the Else block is executed (if it exists).

If, Else, ElseIf, 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 } }
{ Default
  { instructions } }
EndSwitch
```

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

The “Default” blocks are executed if no previous “Case” block was executed. It is possible to use multiple “Default”s, whereby all preceding Cases are regarded, including those before previous Defaults (of course, inside of the same Switch structure). Usually, best use is to have only one Default after all Case blocks.

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.

You can use different value types for each value, the switch expression's result will be converted to the value's type for comparison.

However, floating point values are a bit difficult for two reasons: 1st, there could be rounding troubles. Two seemingly identical values might differ somewhere after e.g. 15 decimals. 2nd, if you enter something like “Case(2)”, it will be valid for 1.5 and 2.4, too, because 2 is an integer and thus the result is converted and rounded. You'd need to use “Case(2**.**)” for float comparisons.

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 } }
{ Default
  { 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.

If an array is used for values, all elements with numerical indexes from 1 to the first unassigned element will be displayed, with a maximum of 256 elements.

Apart from that, it works similar to “Switch”.

In theory, you could also use `Switch( Choice(...) )` (Choice as function, see 9.20.6 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
```

See also 9.20.9 Set entry size and font for choices (SetChoiceEntryFormat)

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:

8.6.1 Looping over data values (list of expressions, array contents, splitted strings, characters of a string)

```
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 key, value in array ( array variable )
```

This loops over all elements in the array, setting the index to the first variable and the value to the second.

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

Similar to the Split function (see 9.5.3 Spit a 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).

8.6.2 Looping over INI file values (sections, values of a section)

ForEach *variable* **in iniSections (** *file name* [, *codepage*] **)**

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

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

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.

For *codepage* parameter, see 9.13.1 Reading a text file (ReadFile, ReadLine).

8.6.3 Looping over registry entries (subkeys, values of a key)

ForEach *variable* **in regSubkeys (** *root, key* **)**

Returns all subkeys of the given key. See 9.19.1 Reading registry entries (RegRead) for parameters.

ForEach *value, data* **in regValues (** *root, key* **)**

Assigns the values of registry key to the given variables.
„value“ is the name of the variable that will receive the value's name, data will receive its current data. Read 9.19.1 Reading registry entries (RegRead) for more informations.
This loop will not return the key's default value (usually shown as “(Default)” or “@” in registry editors).

8.6.4 Looping over files and directories

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"`.

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 Simple iteration (For)

```
For variable = start to end [ step step ]
  { instructions }
Next
```

In the first iteration, *variable* is set to *start*, then it's increased (or decreased, if *step* is negative) by *step* in every further iteration, until *end* is exceeded (*variable*'s value bigger than *end* if *step* is positive, smaller if *step* is negative).

If *step* is omitted, it's 1 if *end* is bigger than *start* or -1 if *end* is smaller than *start*.

This works with both integers and float values. For float values, a precision of 6 digits is used for the end value comparison (see 9.4.8 Compare float values (CompareFloat), too).

Please note that any expressions (for *start*, *end*, or *step*) are only evaluated at the first iteration. For example, if you use a variable for end or step ("For i = 1 to end step x"), and modify this variable during the iteration, the given variable ("i") will still be increased and checked against the values the variables had before "For".

8.9 Break and Continue

```
Break[ ( structure type ) ]
Continue[ ( structure type ) ]
```

With Break, a control structure will be aborted, i.e., the script continues after the correspondinge end statement (EndWhile, Next, ...). This is possible for Switch, Choice, ChoiceDefault, While, ForEach, Repeat, For, and Try.

With Continue, the following code in the same block is skipped and the script continues with the next iteration. If there's no next iteration, the execution continues after the structure's end, just like with Break. This is possible for all loops, i.e. While, ForEach, Repeat, and For. For Try, Continue is allowed as well, but is handled slightly different, see 8.10 Blocks with error handling (Try, Catch).

By default, the innermost control structure's aborted resp. continued.

Example:

```
While( someCondition )
  Switch( value )
    Case( 1 )
      Continue
    Default
      Break
  EndSwitch
EndWhile
```

In this case, “Break” does nothing, because it relates to the Switch statement, and would only skip following lines in the Default block. (Of course, it usually makes more sense in an If statement.)

The “Continue” relates to the While loop, because there's no “Continue” for Switch. This means, all statements after EndSwitch would be skipped as well, and the next iteration will start.

Optionally you can pass a parameter, which structure should be regarded by Break or Continue. In the example, the entire While loop could be aborted with Break(BLOCK_WHILE) instead of just Break. Allowed parameters are: BLOCK_FOR, BLOCK_FOREACH, BLOCK_REPEAT, and BLOCK_TRY. Only for Break additionally: BLOCK_SWITCH and BLOCK_CHOICE (for ChoiceDefault as well).

It's not possible to relate to an outer structure of the same type. E.g., if there's a While loop in another While loop, you can't Break the outer While.

8.10 Blocks with error handling (Try, Catch)

```
Try
  instructions
{ Catch
  instructions }
EndTry
```

This somewhat simplyfies error handling.

The principle is quite simple: If “Break” is invoked in the instructions after Try, the instructions after Catch are immediately executed. Usually the catch block contains some error output (message or logging) and stops execution (Exit) or returns an error flag (ExitSub, more about functions follows later).

With “Continue” in the instructions after Try, execution continues after EndTry. This is handy if for example you try to fill a value by multiple means and like to stop after one did work.

Example:

```
Try
  x = IniRead( "file.ini", "Test", "x" )
  If ( x ne "" )
    Continue
  EndIf

  x = RegRead( HKCU, "Software\Test", "x" )
  If ( x ne "" )
    Continue
  EndIf

  x = Input( "Input text" )
  If ( x eq "" )
    Break
  EndIf
Catch
  Message( "x was neither read nor entered" )
EndTry
```

8.11 Sub routines (Sub, Call/CallFunction, @...)

```
Sub subroutine [ ( parameter { , parameter } ) ]
  { instructions }
EndSub

Call( subroutine {, parameter } )
CallFunction( subroutine, variable {, parameter } )

@subroutine( [ parameter { , parameter } ] )
value = @subroutine( [ parameter { , parameter } ] )
```

With “Call”, “CallFunction” or an *@subroutine*() command/function, the script will continue at the line following the “Sub” with the given subroutine name.

When the end of the subroutine (EndSub or ExitSub) is reached, execution continues where it was invoked. In case of *@subroutine*() functions, this might be in the middle of a parameter expression.

Unlike most other commands, for “Sub” the subroutine name must be given without parentheses and expressions are not allowed. Also, the parameters here are not parameters you pass, but variable names for parameters the sub receives by Call, CallFunction, or *@subroutine*().

For Call/CallFunction, please be aware you have to use quotes around the subroutine name, since it's an expression like every other parameter. You might even use something like “Call(variableContainingMySubroutine)” 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.

For Call, the old syntax style (“Call SubFunction”) might be handy, too, but it's only recommended if you don't use parameters. Better just use “@SubFunction()”.

If you pass parameters, the subroutine will have two local variables (see 7.5.2 Variable scope) defined: argc contains the number of passed parameters, argv is an array with all passed parameters. So if you pass two parameters, argc is 2, argv[1] the first parameter, and argv[2] the second.

If parameters are defined after the sub name, those will be defined as local variables. If there are less parameters than target variables, the non-passed parameters will be empty (IsEmpty()). If there are more parameters than target variables, the rest (only that!) will be put to argc/argv.

If you use CallFunction, a value set with “Return(value)” or “ExitSub(value)” in the subroutine is set to the given variable. If you use @subroutine() in an expression, the return value's used just like with MortScript's internal functions (like e.g. Input or Length).

Mind that Return() doesn't leave the subroutine like in many other languages, it only sets the return value! If Return() is not invoked, the variable will be empty (see 9.2.4 Check if variable is defined (IsEmpty)). Also consider ExitSub (see 8.15 Leave subroutine (ExitSub)).

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

Examples:

```
@MySub( "x" )
y = @MySub( 1, 2, 3 )
Call( "Other" & "Sub", @MySub(1,2) + 2 )
CallFunction( "My" & "Sub", result, 1 )

Sub MySub( p1, p2 )
  If ( IsEmpty( p1 ) OR IsEmpty( p2 ) )
    Message( "Not enough parameters passed" )
    ExitSub( 0 )
  EndIf
  If ( argc > 0 )
    Message( "More than two parameters passed" )
  EndIf
  Return( p1 + p2 )
EndSub

Sub OtherSub
  Message( argv[1] )
EndSub
```

8.12 Include sub routines of other files (Include)

`Include(` *`file`* **`)`**

With Include, all Sub blocks of the given file will be included as if they were defined in the current script.

The file embedded with Include may contain other Include commands itself, but shouldn't contain any other commands outside of the Sub blocks. (They'd be simply ignored)

Include commands should be at the beginning of the script. They're interpreted before the script is executed, because otherwise Call commands might not find the required routine. Due to this, it's also not possible to use variables for the file parameter.

Please regard that with many Includes there's an increased risk you use the same Sub name twice. In this case, MortScript will exit with an error message before executing the script.

8.13 Other script as subroutine (CallScript/CallScriptFunction)

CallScript(*MortScript file* {, *parameter* } **)**
CallScriptFunction(*MortScript file*, *variable* {, *parameter* } **)**

Executes the given script as if it were a subroutine.

This goes also for variables, i.e., it might modify variables of the invoking script! So you should invoke Local() or Global() in the called script.

Parameters (argc and argv) and return values (Return()) work like in Sub routines, too. See 8.9 Sub routines (Sub, Call/CallFunction) for more about that.

Please regard 6.5 Directories and files, too.

CallScript(Function) was one of the first ways to allow something like subroutines. You should prefer Include and Call(Function) / @subroutine() nowadays.

Example:
```
CallScript( "subscript.mscr" )
```

To execute other applications or „independent" scripts, there are own commands. see 9.6 Execute applications or open documents.

8.14 Set return value (Return)

Return(value **)**

Returns the given value to an invoking CallFunction or CallScriptFunction call.

Unlike most other languages, this will not leave the Sub routine resp. invoked script, it only sets the return value! Arrays are allowed.

See also 8.9 Sub routines (Sub, Call/CallFunction) and 8.10 Other script as subroutine (CallScript/CallScriptFunction).

8.15 Leave subroutine (ExitSub)

ExitSub[**(** *value* **)**]

Leaves the current sub routine. If a value is passed, it's returned like with Return(...). Otherwise, either a value set with Return or "nothing" (IsEmpty()) is returned.

8.16 Abort script (Exit)

Exit

Stops executing the script.

9 Commands and functions

Functions are represented by *type* `= Function( ... )`.

The type is either *string, bool, int,* or *float*, depending on the type of the return value. *int* and *bool* both mean integer numbers (no decimals), but *bool* functions will only return TRUE (1) and FALSE (0).

If different types might be returned, it's just *value*. (Usually more informations are given in the description)

Of course, all functions can be used in more complex expressions than a simple “variable = ...” assignment, 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 Variables

9.2.1 Assigning variables ("=", "+=", ... and Set)

```
Variable = expression
Variable += expression
Variable -= expression
Variable *= expression
Variable /= expression
Variable &= expression
Variable \= expression
Set( variable, expression )
```

Assigns the result of the expression to the variable.

Set() shouldn't be used anymore, better use *Variable = expression*.

The combined assignment operators (+= and the like) are shortcuts for *variable = variable operator expression*. I.e. `i+=1` is the same as `i=i+1`.

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 containing no %s is found (or the system crashes with a stack overflow...)

This usually is quite confusing, error prone and a bit outdated. Please use references (see 7.5.4 References ([variable name])) instead wherever possible and necessary.

9.2.2 Expressions in a string (Eval)

```
value = 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

9.2.3 Remove variable or array element (Clear)

```
Clear( variable )
```

Clear will remove the variable or array element. Contrary to setting the variable or array element to an empty string, IsEmpty() will return TRUE (see below) and an array element won't be there in ForEach, Choice, etc.

9.2.4 Check if variable is assigned (IsEmpty)

```
bool = IsEmpty( variable )
```

Returns TRUE if the variable (or array element) wasn't assigned so far or removed with Clear(), FALSE if the variable's assigned – even if it's just an empty string.

9.2.5 Typ einer Variablen bestimmen (VarType)

```
int = VarType( variable )
```

Returns the type of a given variable.

Possible return values:

- VAR_EMPTY
- VAR_INT
- VAR_FLOAT
- VAR_STRING
- VAR_ARRAY
- VAR_WINDOW

9.2.6 Variable scope (Local, Global)

```
Local( [ variable {, variable } ] )
Global( variable {, variable } )
```

With Local(), the given or all (if no variables are passed) variables will be local to the current block (subroutine or main script function).

Global() works the other way around: The given variables are accessed globally, while everything else is local to the current block.

See 7.5.2 Variable scope for more informations.

9.3 String operations

9.3.1 Get the length of a string (Length)

```
int = Length( string )
```

Returns the number of characters in a string.

Example:
```
x = Length( "This is a test" )
```
→ x = 14

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

string = **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"

9.3.3 Einzelnes Zeichen einer Zeichenfolge (CharAt)

string = **CharAt(** *string*, *position* **)**

Returns the character at the given position. If the string doesn't contain enough characters, „nothing“ is returned (see 9.2.4 Check if variable is assigned (IsEmpty)).

9.3.4 Split string and return a single part (Part)

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

Splits the string on each occurrence of the separator, 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 TRUE or omitted, any spaces surrounding the part will be removed.

See also 9.5.3 Spit a string to multiple variables/array elements (Split)

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( "one, two, three", ",", 4 )
```
→ x = "" (empty string for not existing parts)

9.3.5 Find a string in another string (Find)

`string = Find( string to check, string to search [, start ] )`

Returns the position of the first character where the searched string is found.

If a *start* position is given, searching starts from that position.

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( "abcdefcd", "cd", 5 )
```
→ x = 7

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

9.3.6 Find last occurrence of a character (ReverseFind)

`int = ReverseFind( string, character )`

Returns the position of the last occurrence of “character” in the “string”. Unlike 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 Replace strings (Replace)

`string = Replace( source, old, new )`

Replaces all occurrences of “old” in the “source” string with the “new” string.

Example:
```
x = Replace( "My old string", "old", "new" )
```
→ x = `"My new string"`

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

```
string = ToUpper( string )
string = 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 (unlike 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"

9.3.9 Covert character to/from Unicode value (UcChar, UcValue)

```
string = UcChar( value )
int = UcValue( string )
```

UcChar returns the charater for a given Unicode value, UcValue returns the value for a given charater (single character string as parameter, otherwise only the first character is regarded!).

Examples:
```
x = UcValue( "A" )
```
→ x = 65

```
c = UcChar( x + 1 )
```
→ c = "B"

9.3.10 Parts of a filename (FilePath, FileBase, FileExt)

```
string = FilePath( file with path )
string = FileBase( file with path )
string = FileExt( file with path )
```

These functions help to split a filename with path.

FilePath returns the path without the filename ("\My documents" for "\My documents\test.txt”)

FileBase returns the filename without path and extension ("test" for "\My documents\test.txt”)

FileExt returns the extension ("txt" for "\My documents\test.txt”)

Might be useful with 9.18.4 Getting system paths (SystemPath).

9.4 Math functions

9.4.1 Formatted output (Format)

```
string = Format( value, decimals )
```

Returns the value as string with the given precision. The value will be rounded commercially if necessary.

Examples:
```
x = Format( 123.456789, 2 )
```
→ x = "123.46"

```
x = Format( 12, 2 )
```
→ x = "12.00"

9.4.2 Conversion to/from hexadecimal (NumberToHex, HexToNumber)

```
string = NumberToHex( int value )
int = HexToNumber( string )
```

NumberToHex converts an integer value (floats are rounded) to a string with the hexadecimal value. The string is formatted to use full bytes, i.e. an even number of characters (like "0100" for 256 or "0a" for 11).

HexToNumber works the other way: It converts a string with an hexadecimal value to its integer value. It will work to the first invalid character, e.g. HexToNumber("axe") will return 10, because "a" is a valid hex digit. Is also allows uppercase ("ADE").

Please note these functions will only work reliable with numbers from 0 to 2,147,483,647 (7fffffff). Numbers from 2,147,483,648 (80000000) to 4,294,967,295 (ffffffff) will be converted fine from decimal to hexadecimal, but will return negative values when converted from hexadecimal to decimal. Also, -1 to -2,147,483,647 will return 8 byte hex values from ffffffff (-1) to 80000001 (-2,147,483,647). That's because of the way negative values are stored internally (the first bit is the "negative" flag).

Values exceeding this range will cause errors or strange results.

9.4.3 Rounding (Round, Floor, Ceil)

```
int/float = Round( value [, precision] )
int/float = Floor( value [, precision] )
int/float = Ceil( value [, precision] )
```

These functions return the rounded value as integer if precision is 0 or omitted.

If a precision is given, it returns a float rounded to the given number of decimals, e.g. Round(2.56, 1) will return 2.6.

Floor rounds down (2.9 → 2), Ceil rounds up (2.1 → 3), and Round uses commercial rounding, i.e. rounding up starting with .5 (2.49 → 2, 2.5 → 3).

9.4.4 Random values (Rand)

```
int = Rand( max )
float = Rand()
```

If a max parameter is given, it returns integer values from 0 to max-1.

If no parameter is given, it returns float values from 0 to 0.999...

9.4.5 Trigonometric functions (Sin, Cos, Tan, etc.)

```
float = Sin( radians )
float = Cos( radians )
float = Tan( radians )
float = SinH( radians )
float = CosH( radians )
float = TanH( radians )
float = ArcSin( radians )
float = ArcCos( radians )
float = ArcTan( radians )
```

ArcSin, -Cos, and -Tan compute the arcus sinus, cosinus, resp. tangens; SinH, CosH, and TanH the hyberbolic one.

Please note these functions use the radians as parameter! If you've got the degree (like 45°), you can convert it with “degree * PI / 180”.

9.4.6 Logarithms and exponentials (Log, Log10, Exp)

```
float = Log( value )
float = Log10( value )
float = Exp( value )
```

Log calculates the logarithm based on *e* (MortScript variable EULERT), Log10 based on 10.

Exp(value) is identical to EULERT ^ value.

9.4.7 Square root (Sqrt)

```
float = Sqrt( value )
```

Calculates the square root of the given value. The square root of 2 is also available in the variable SQRT2.

9.4.8 Compare float values (CompareFloat)

```
int = CompareFloat( value1, value2, precision )
```

Comparing float values is a bit tricky because there can be rounding errors. A zero might not be a “real” zero, but something like $1*10^{-20}$.

With this little helper function, the values are rounded to the given precision and then compared.

It returns -1 if value1 < value2, 1 if value1 > value2, and 0 if both values are equal.

9.4.9 Get biggest/smallest value (Min/Max)

```
value = Min( value, value {, value } )
value = Max( value, value {, value } )
```

Returns the biggest (Max) resp. smallest (Min) value of the parameter list (at least 2 parameters). The values are compared numerically, the return value will be the one of the corresponding parameter. I.e., Max(1, 2.5, "3.33") will return the string "3.33".

9.5 Arrays

9.5.1 Get biggest index in series (MaxIndex)

int = **MaxIndex(** *array* **)**

Returns the biggest numerical index defined in a row starting from 1, maximum value is 256.

Example:
```
array[1]="a"
array["2"]="b"
array[3]="c"
array[5]="e"
array["x"]="X"
max = MaxIndex[array]
```

will return 3.

MortScript converts string indexes with numerical contents to numerical indexes unless they start with 0, so "2" is regarded, while "x" isn't. See also 7.5.3 Arrays (Lists).

Because 4 is missing, 5 will be ignored. It would be the same if array[4] was defined and cleared with Clear() afterwards.

9.5.2 Get number of elements (ElementCount)

int = **ElementCount(** *array* **)**

Contrary to MaxIndex, this returns the count of all array elements that were ever assigned, including those “removed” with Clear() (which only removes the element's value, but not the element itself).

For the array in the MaxIndex example, ElementCount would return 5.

9.5.3 Create an array from a list of values (Array)

array = **Array(** *value* {, *value* } **)**

Returns an array with the given values, indexes start with 1.

Please note MortScript isn't able to do something like “Array("a", "b", "c")[1]”, you can only assign an array variable and then access the elements of that variable.

Example:
```
days = Array( "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" )
day = days[ FormatTime("w")+1 ]
```

9.5.4 Create an array with named indexes (Map)

```
array = Map( index, value {, index, value } )
```

Returns an array with the given keys and values.

You must pass an even number of parameters, whereby the first value of each pair is the index and the second is the value.

Please note MortScript isn't able to do something like “Map(1, "a", 2, "b")[1]”, you can only assign an array variable and then access the elements of that variable.

Example:
```
months = Map( "01", "Jan", "02", "Feb", "03", "Mar", etc. )
month = months[ FormatTime("m") ]
```

9.5.5 Spit a string to multiple variables/array elements (Split)

```
Split( string, separator, trim?, variable {, variable } )
array = Split( string, separator [, trim?] )
```

Splits the string on each occurrence of the separator. If the separator consists of multiple characters, they must occur that way in the string which is splitted.

If you use the command version with multiple variables, the parts are assigned to 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 only a single variable is given or the function is used, the assigned variable will be an array with the single parts, i.e., variable[1] to variable[*n*]. Each assigned part will be a string value.

If "trim?" is TRUE (or omitted in the function), any spaces surrounding the parts will be removed.

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"

```
list = Split( "one, two, three", "," )
```
→ list[1]="one", list[2]="two", list[3]="three"

9.5.6 Join multiple array elements to string (Join)

```
string = Join( array [, concatenation ] )
```

Join is the counterpart to “Split”. It concatenates all elements with numerical index starting from 1 to the first unassigned index to a single string. If *concatenation* is given, this string will be inserted between the single elements.

Examples:
```
str = Join( Array( "a", "b", "c" ), "|" )
```
→ str = "a|b|c"

```
arr[1] = "This"
arr[2] = "is"
arr["3"] = "a"
arr[04] = "test"
arr["test"] = "of"
arr[6] = "Join"
str = Join( arr )
```
→ str = "Thisisatest" ("3" is converted to numerical index, "test" is alphanumerical, 5 is missing, so 6 isn't regarded)

9.5.7 Check for existence of array element (Join)

```
bool = InArray( array [, value ] )
```

Checks whether the given *value* is contained in the values of the given *array*. Checking the existence of an index can be done with IsEmpty(array[index]).

The comparison is done similar to Switch (8.3), i.e. the data type of the value decides how the values are compared.

9.6 Execute applications or open documents

9.6.1 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" )
```

9.6.2 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.

9.6.3 Other script as sub routine (CallScript, CallScriptFunction)

CallScript(*MortScript file* {, *parameters* } **)**
CallScriptFunction(*MortScript file, variable,* {, *parameters* } **)**

Executes another script as if it were a subroutine.

See also 8.9 Other script as subroutine (CallScript)

9.6.4 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" )
```

9.6.5 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

9.6.6 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

9.6.7 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

9.7 Application windows

9.7.1 Window titles and variables – how MortScript finds a window

In many commands and functions of this section (and some others, like most Send... commands), a window must be passed.

If a window title is passed as string, MortScript looks for a window which contains the given text as title. This is done case sensitive, i.e. “Word” will not find “WORD”. If there are mulitple windows, MortScript tries a “best fit”, regarding the following (top = most important):

- Is the window a main window? (it has no parent, or the parent is the desktop)
- Is the given text found at the beginning of the title?
- Is the window visible? (that's not the same as “in the background”, it's more like “not shown in task list”)

As an alternative, it's possible to pass a special “window value” instead of the title.

This special value is returned by the functions ActiveWindow(), FindWindow(), and FindWindows().

If such a variable (or direct return value) is used where a string is expected, the window title is used (e.g. in Message(ActiveWindow())). Additionally, a “window handle” is stored internally, which allows MortScript to access the desired window faster and without risk of confusing it with another window with the same or similar title.

If you use operations (e.g. window = window & " Document1"), this information will be lost.

9.7.2 Show and activate a window (Show)

Show(*window* **)**

Activates the window with the given title.

9.7.3 Minimize/hide a window (Minimize)

Minimize(*window* **)**

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

9.7.4 Close a window / end application (Close)

Close(*window* **)**

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.

9.7.5 Get the currently active window (ActiveWindow)

window = **ActiveWindow()**

Returns the currently active window (title and handle, see 9.7.1 Window titles and variables – how MortScript finds a window).

9.7.6 Check whether a window is active (WndActive)

```
bool = WndActive( window )
```

Returns TRUE, if a window with the given window is active, otherwise FALSE.

9.7.7 Search windows (FindWindow, FindWindows)

```
window = FindWindow( title [, position [, case sensitive?
                    [, only apps? [, invisible?
                    [, class ]]]]] )

array = FindWindows( [ title [, position [, case sensitive?
                    [, only apps? [, invisible?
                    [, class ]]]]]] )
```

Looks for the window resp. the windows with the given properties:

- *Title*: The window title or a part of it
- *Position*: One of the following predefined constants (default: ANYWHERE):
  - ➔ ANYWHERE = *Title* appears anywhere in window title
  - ➔ BEGINNING = *Title* appears at the beginning of the window title
  - ➔ COMPLETE = *Title* contains the entire window title
- *Case sensitive?*: Regard upper/lower case for comparison? (default: no)
- *Only apps?*: Regards only windows which have some properties common for "main windows", like having no parent or the desktop as parent. This omits e.g. dialog elements, which are separate windows in Windows as well. (default: no)
- *invisible?*: Also include invisible windows? (default: no)
- *class*: You can enter a window class here. Comparison is always complete and case sensitive.

FindWindow returns only a single window. If there are multiple fittings, an internal priority is used. For example, a window with the search title at the beginning comes before one where it's somewhere in the middle, or an app window is preferred to child windows. The window title is mandatory because otherwise there would be too many possible windows. If no window is found, an empty value is returned (see 9.2.4 Check if variable is assigned (IsEmpty)).

FindWindows returns all found windows as array. This is also the case if only one window (single element) or no window (empty array) is found.

9.7.8 Check whether a window exists (WndExists)

```
bool = WndExists( window )
```

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

9.7.9 Wait until a window exists (WaitFor)

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

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

9.7.10 Wait until a window becomes active (WaitForActive)

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

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

9.7.11 Get window title / element contents (WindowText)

```
string = 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.

9.7.12 Get window class (WindowClass)

```
string = WndClass( window )
```

Returns the class of the given window. It's an internal label which is assigned by Windows or the application, e.g.. „Dialog“, „Edit“, etc.

9.7.13 Get window position (GetWindowPos, WndLeft, -Right, -Top, -Bottom)

```
GetWindowPos( window, left, top, right, bottom )
int = WndLeft( window )
int = WndRight( window )
int = WndTop( window )
int = WndBottom( window )
```

Receives the position of the named window.

GetWindowPos sets the variables given for left, top, right, and bottom.

The functions return a single border position each.

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

```
SendCommand( [ window, ] command id )
SendOK [ ( window ) ]
SendCancel [ ( window ) ]
SendYes [ ( window ) ]
SendNo [ ( window ) ]
```

With these commands, pressing the corresponding button is emulated.

If no window title is given, the currently active window is used.

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

9.7.15 Send advanced commands/messages (SendCommand, SendMessage, PostMessage)

```
SendCommand( [ window, ] command id )
PostMessage( [ window, ] message id, wparam, lparam )
SendMessage( [ window, ] message id, wparam, lparam )
int = SendMessage( [ window, ] message id, wparam, lparam )
```

SendCommand allows you to send any command id you want (usually all buttons and menu entries), but this'll require good connections to the programmer, because the command ids are different for every program and might even be different after an update.

Same goes for SendMessage and PostMessage, which are a more generic approach. You need good connections to the application programmer to use those, it could also cause big troubles (usually program crashes) if unexpected data is sent. SendMessage causes the message to be handled immediately, and thus also allows to get a return value. With PostMessage, the message is added to the message queue, so your script continues immediately, while the message is handled some time later. MortScript only supports numerical parameters.

9.8 Keystrokes

9.8.1 Sending strings (SendKeys)

SendKeys([*window,*] *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" )
```

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

SendSpecial(*key name* [, *state*]] **)**

Emulates the given keystroke.

Currently available keys are: Alt, Ctrl, Shift, CR, Win, Context, Tab, ESC, Space, Up, Down, Left, Right, Home, End, PageUp, PageDown, Delete, Backspace, Insert, Snapshot, F1 - F12, LeftSoft, RightSoft.

The key's name must be passed as string. It's not case sensitive (i.e. “Esc” and “ESC” are the same). Not all keys work on all systems, e.g. the soft keys are unknown to the desktop.

It's also possible to pass a numerical scan code. This is highly device dependent (esp. the localization of the system) and is more something for “advanced hackers”.

If *state* is omitted, the key is pressed for a short time and then released. You can use the states “down” and “up” for keeping the key pressed until you release it again (handy for Alt, Ctrl, Shift, and Win). Don't forget the “up”, otherwise it might cause some confusion afterwards...

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

Activates the given window and sends the given special character. If no window title is given, the currently active window is used.
Ctrl?, Shift?, and *Alt?* are switches for the corresponding keys. If the parameter is TRUE, 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)

9.8.3 Copy screen contents to clipboard (Snapshot)

`Snapshot [ ( `*`window`*` ) ]`

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).

9.8.4 Sending Ctrl+key (SendCtrlKey)

`SendCtrlKey ( [ `*`window`*`, ] `*`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).

9.9 Mouse clicks / tapping

9.9.1 Single click (MouseClick)

```
MouseClick( [ window, ] x, y )
RightMouseClick( [ window, ] x, y )
MiddleMouseClick( [ window, ] 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.

9.9.2 Double click (MouseDblClick)

```
MouseDblClick( [ window, ] x, y )
RightMouseDblClick( [ window, ] x, y )
MiddleMouseDblClick( [ window, ] 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.

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

```
MouseDown( [ window, ] x, y )
MouseUp( [ window, ] x, y )
RightMouseDown( [ window, ] x, y )
RightMouseUp( [ window, ] x, y )
MiddleMouseDown( [ window, ] x, y )
MiddleMouseUp( [ window, ] 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.

9.10 Waiting

9.10.1 Fixed delay in milliseconds (Sleep)

Sleep(*milliseconds* **)**

Waits the specified time.

9.10.2 Wait message with countdown / condition (SleepMessage)

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

Shows a wait message with a countdown.

If *OK allowed?* is TRUE, 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.

See also 9.20.10 Set font for big messages (SetMessageFont)

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

9.10.3 Waiting for windows (WaitFor / WaitForActive)

See 9.7.10 Wait until a window exists (WaitFor) and 9.6.11 Wait until a window becomes active (WaitForActive)

9.11 Time

9.11.1 Unix timestamp (TimeStamp)

```
int = TimeStamp()
int = MakeTimeStamp( year, month, day
                     [, hour [, minute [, second ]]] )
```

Returns the current time in seconds since 01/01/1970 (aka Epoch).

TimeStamp() returns the value for the current system time, MakeTimeStamp(...) for the given date/time combination. If you omit time parameters, “0” is assumed, so if you omit all three of them, midnight is used.

9.11.2 Formatted output (FormatTime)

```
string = 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
{MM}	Month name (e.g. “January”)
{M}	Month name abbreviated (e.g. “Jan”)
{WW}	Day of week name (e.g. “Monday”)
{W}	Day of week name abbreviated (e.g. “Mon”)

All other characters remain unchanged.

Note all return values will be strings. This is to allow leading zeroes, like "02" for february, which is handy to combine filenames. However, it might cause troubles when using arrays. You either need to assign the array elements with strings (“Month["01"] = "First"”) or convert the string to a number, e.g. by using “FormatTime("m")*1”.

Examples:

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

9.11.3 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).

Note all variable values will be strings. This is to allow leading zeroes, like "02" for february, which is handy to combine filenames.

9.11.4 Set current time (SetTime, SetDate)

```
SetTime( hour, minute, second [, day, month, year ] )
SetDate( day, month, year )
```

Sets the system time to the given time resp. date. With SetDate, the time will not be modified.

Hint: Please don't use this command to bypass evaluation periods. Be honest and pay for the programs you're using or try to find free alternatives.

9.12 Copy, rename, move, and delete files

9.12.1 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 FALSE or omitted, already existing files won't be overwritten.

Example:

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

9.12.2 Copy multiple files (XCopy)

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

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 FALSE or omitted, already existing files won't be overwritten.

If *subdirs?* is TRUE, all files fitting to the source filter contained in subdirectories will be copied, too. The target subdirectories are created if necessary.

Examples:

```
XCopy( "\My documents\*.txt", "\Storage" )
XCopy( "\My documents\*.txt", "\Storage", TRUE, TRUE )
```
(will copy also "\My documents\texts\x.txt" to "\Storage\texts\x.txt")

9.12.3 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 FALSE or omitted, already existing files won't be overwritten.

9.12.4 Move multiple files (Move)

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

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 FALSE or omitted, already existing files won't be overwritten.

If *subdirs?* is TRUE, all files fitting to the source filter contained in subdirectories will be moved, too. The target subdirectories are created if necessary. The source directories will not be removed if they're empty afterwards.

9.12.5 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").

9.12.6 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.

If no file filter is given (e.g. `DelTree( "\Temp" )`) and the given path exists, *.* is assumed.

Please handle with care!

9.12.7 Creating a shortcut/link (CreateShortcut)

```
CreateShortcut( shortcut file, target file [, overwrite? ] )
```

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

If *overwrite?* is FALSE or omitted, already existing files won't be overwritten.

Example:
CreateShortcut("\Windows\Start Menu\Test.lnk","\Storage\Test.exe")

9.13 Reading and writing text files

9.13.1 Reading a text file (ReadFile, ReadLine)

```
string = ReadFile( file name [, length [, codepage ] ] )
string = ReadLine( file name [, codepage ] )
```

ReadFile reads the entire contents of a text file into the variable. The file size is limited to *length*, 1 MB or the available memory (whichever is the least). If *length* is 0, the default maximum (1 MB) is used.

ReadLine reads a single line of the given text file. The next time ReadLine is invoked, it returns the next line, etc. If there are no more lines in the file, it returns an empty value (see 9.2.4 Check if variable is assigned (IsEmpty)). Opposed to ReadFile, ReadLine only works with “real” files, not with serial ports (see 9.13.6 Access serial ports (SetComInfo)) or Internet access.

Possible values for the codepage are either the codepage numbers (if you know them) like 1252 (Western Europe), 437 (American DOS), ... or any of those strings:

- "latin1" (Western Europe)
- "jis" (Japanese)
- "wansung" (Korean)
- "johab" (Korean)
- "chinesesimp" (Chinese simplified)
- "chinesetrad" (Chinese traditional)
- "hebrew"
- "arabic"
- "greek"
- "turkish"
- "baltic"
- "latin2" (mostly Eastern Europe)
- "cyrillic"
- "thai"
- "utf8" (special encoding only for non-ASCII characters)
- "unicode" (2 bytes for each character, to be more precise UTF-16 little endian)
- "utf8-prefix" (like "utf8", but file starts with the hex values EF BB BF, which is an indicator for some editors/programs)
- "unicode-prefix" (similar to "utf8-prefix", but with FF FE prefix and unicode)

Default is the system's default codepage, which depends on your Windows localization. UTF8 or Unicode encoded files are also recognized, if they start with the corresponding prefixes (see “-prefix” encodings above).

You can parse the file e.g. with ForEach line in split (contents, "^LF^", TRUE)

See also the ForEach possibilities for INI files and ReadINI/WriteINI!

Example for ReadLine:

```
StatusType( ST_LIST, TRUE, FALSE )
StatusHistorySize( 500 )

line = ReadLine( "test.txt" )
While( NOT IsEmpty( line ) )
  StatusMessage( line )
  line = ReadLine( "test.txt" )
EndWhile
```

9.13.2 Writing to a text file (WriteFile, WriteLine)

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

Writes the contents to the file.

For WriteFile, if *append?* is FALSE or the parameter is omitted, an existing file will be overwritten, otherwise the given contents is appended at the end of the existing data.

WriteLine always appends at the end, and also adds a new line. The file remains opened until CloseFile or the end of the script. Due to that, WriteLine is faster than WriteFile with append option, but there might be problems when the file is accessed otherwise (other programs, trying to delete it, …).

For possible values for codepage, please see ReadFile above. The “-prefix” variations only are applied if *append?* is FALSE!

9.13.3 Closing a file (CloseFile)

CloseFile(*file name* **)**

Closes a file which was opened with ReadLine or WriteLine. ReadLine automatically closes a file after the last line. All opened files are closed after the script ends. Nonetheless, you should close files whenever you don't need to access them anymore to avoid access problems.

9.13.4 Reading a value of an INI file (IniRead)

string = **IniRead(** *file name*, *section*, *entry* [, *codepage*] **)**

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

For *codepage* parameter, see 9.13.1 Reading a text file (ReadFile, ReadLine).

Example:

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

9.13.5 Writing a value to an INI file (IniWrite)

```
IniWrite( file name, section, entry, value [, codepage ] )
```

Writes an entry to an INI file. The section name must be passed without the brackets.

For *codepage* parameter, see 9.13.1 Reading a text file (ReadFile, ReadLine).

Be aware this causes MortScript to load, parse, and write the entire file. It might be better to do this yourself (ReadFile, ForEach with split, WriteFile) if many values are modified.

Example:
```
IniWrite( "\My documents\test.ini", "Settings", "Test", "x" )
```

9.13.6 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.

You should use ReadFile with a maximum size, otherwise there might be a huge lag until the timeout is reached. E.g. `data = ReadFile( "COM1:", 100 )`.

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.

9.14 File system informations

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

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

Returns TRUE, if the file or directory exists, FALSE if not.

It also returns FALSE, if the entry doesn't correspond to the queried type. This means, "FileExists("\Windows")" is FALSE, because it's a directory, not a file.

9.14.2 Check free space (FreeDiskSpace)

```
int = FreeDiskSpace( directory [, unit] )
```

Returns the free disk space in the given directory. By default, it's in bytes, you can specify another unit by passing BYTES, KB, MB, or GB as unit parameter (constants, i.e. without quotes). The maximum return value is 2147483147, which is about 2GB for size in bytes, 2TB for KB, etc.

On Windows Mobile devices, the directory (e.g. "\Storage" for the storage card), on PCs the drive letter ("D:\...") is regarded.

9.14.3 Check disk size (TotalDiskSpace)

```
int = TotalDiskSpace( directory [, unit] )
```

Returns the size of the disk of the given directory. By default, it's in bytes, you can specify another unit by passing BYTES, KB, MB, or GB as unit parameter (constants, i.e. without quotes). The maximum return value is 2147483147, which is about 2GB for size in bytes, 2TB for KB, etc.

On Windows Mobile devices, the directory (e.g. "\Storage" for the storage card), on PCs the drive letter ("D:\...") is regarded.

I know, 2GB isn't much, but handling larger numbers would be quite complicated, because that number is the highest value a 32 bit system can handle without workarounds... (Yes, 4GB would be possible, too, but then negative numbers wouldn't be supported at all in MortScript...)

9.14.4 Get file size (FileSize)

```
int = FileSize( file name [, unit] )
```

Returns the size of the file in bytes. By default, it's in bytes, you can specify another unit by passing BYTES, KB, MB, or GB as unit parameter (constants, i.e. without quotes). The maximum return value is 2147483147, which is about 2GB for size in bytes, 2TB for KB, etc.

9.14.5 Get file creation time (FileCreateTime)

int = **FileCreateTime(** *file name* **)**

Returns the creation time of the file as unix timestamp, or 0 if the file doesn't exist.

See also 9.11 Time for informations about how to compare or format it.

9.14.6 Get file modification time (FileModifyTime)

int = **FileModifyTime(** *file name* **)**

Returns the last modification time of the file as unix timestamp, or 0 if the file doesn't exist.

See also 9.11 Time for informations about how to compare or format it.

9.14.7 Get file attributes (FileAttribs)

bool = **FileAttribute(** *file name*, *attribute* **)**

Returns the current state of the given file attribute. TRUE = attribute is set, FALSE = 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).

9.14.8 Set file attributes (SetFileAttribute, SetFileAttribs)

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

Sets (*set?*=TRUE) resp. removes (*set?*=FALSE) 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).

<u>Examples:</u>

```
SetFileAttribute("\Test.txt", "hidden", TRUE)
```

→ hides the file

```
SetFileAttribute("\Test.txt", "readonly", FALSE)
```

→ removes the write protection

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

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

<u>Examples:</u>

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

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

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

→ removes read only attribute, all other attributes remain unmodified

9.14.9 Get version number (FileVersion / GetVersion)

string = **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 (integer values).

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.

9.14.10 Get files/directories of directory (DirContents)

array = **DirContents(** *files, type* **)**

Returns an array with the files and/or directories contained in the given directory. The elements contain the file resp. directory names without the given directory (e.g. “x.txt”, not “C:\Directory\x.txt”).

The *files* parameter consists of path and a file filter, like "\Windows*.exe".

type can be DC_FILES, DC_DIRS, or DC_ALL (constants, i.e. do not quote!). With DC_FILES, the contained files are returned, with DC_DIRS the directories, and DC_ALL returns both.

9.15 ZIP archives

9.15.1 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). Additionally, Java's ZIP functions use UTF8.

9.15.2 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. Contrary 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" )
```

9.15.3 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 TRUE, 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", TRUE, "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.

9.15.4 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“

9.15.5 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.

9.15.6 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”.

9.16 Connections

9.16.1 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

9.16.2 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.

Contrary 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

9.16.3 Check connection (Connected/InternetConnected)

bool = **Connected()**
bool = **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 TRUE if a connection exists, FALSE 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 TRUE) 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

9.17 Internet access

9.17.1 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

9.17.2 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

9.17.3 Other possibilities

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

9.18 Directories

9.18.1 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.

9.18.2 Remove directory (RmDir)

`RmDir(` *`directory`* **`)`**

Removes the directory.

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

9.18.3 Change directory (ChDir)

`ChDir(` *`directory`* **`)`**

Makes the directory the current directory.

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

9.18.4 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
AppData	”\Application data” or localization
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") )
```

9.19 Registry

9.19.1 Reading registry entries (RegRead, RegReadExt)

value = **RegRead(** *root*, *key*, *value name* **)**
string = **RegReadExt(** *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! Since V4.11, constants are defined, too, so you don't need to use quotes.

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

For RegRead, the value's data type is automatically regarded. DWords are returned as integer number, string values as strings, binary data as a string containing the data as hex dump (e.g. "010ACF"), and “MultiString”s as array with string elements.

RegReadExt works similar to RegRead, but the result is returned as string in the style of Microsoft's .reg files, i.e.:

- dword:00000000 for integer numbers (hexadecimal)
- "..." for strings (including quotes, contained quotes as \")
- hex:00,00,.... for binary data (hex dump)
- hex(7):00,00,... for multiple strings (hex dump)
- hex(2):00,00,... for expandable strings (hex dump)

9.19.2 Writing registry entries (RegWriteString/-DWord/ -Binary/-MultiString, RegWriteExt)

RegWriteString(*root, key, value name, value* **)**
RegWriteDWord(*root, key, value name, value* **)**
RegWriteBinary(*root, key, value name, value* **)**
RegWriteMultiString(*root, key, value name, array* **)**
RegWriteExt(*root, key, value name, value* **)**

Writes a value to the registry.

Valid values for *root* are listed at 9.19.1 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!

RegWriteMultiString writes a list of string values (given as array). The given value must be an array. All elements from 1 to the first unassigned number are regarded, if the elements will be converted to strings if necessary.

RegWriteExt is the counterpart to RegWriteExt (see above), it requires a value in .reg style.

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" )

RegWriteMultiString( "HKCU", "Software\Mort\Dummy", "Days", \
                     Array( "Mon", "Tue", "Wed" ) )


RegWriteExt( "HKCU", "Software\Mort\Dummy", "IntVal", \
                     "dword:0000002A" )
```

9.19.3 Checking existence of a value (RegValueExists)

bool = **RegValueExists(** *root, key, value name* **)**

Returns TRUE, if the given value exists, FALSE if it doesn't.

Values for *root* are as in RegRead.

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

bool = **RegKeyExists(** *root*, *key* **)**

Returns TRUE, if the given key (a “subdirectory” in the registry) exists, FALSE if it doesn't.

Values for *root* are as in RegRead.

9.19.5 Checking the type of a value (RegType)

string = **RegType(** *root*, *key*, *value name* **)**

Returns the type of a registry value.

Possible return values are “binary”, “dword” (integer value), “link” (extremely rarely used), “multi_sz” (multiple strings), “sz” (string), “expand_sz” (string with expandable variables), “none” (no value exists), and “unknown” (new / user defined types).

9.19.6 Removing a registry value (RegDelete)

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

Removes the registry value.

Values for *root* are as in RegRead.

9.19.7 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 TRUE, all contained values are removed, too.

values? is also used for sub keys if *sub keys?* is TRUE.

I.e., RegDeleteKey("HKCU", "\Software\Something", FALSE, TRUE) 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!

9.20 Dialogs

9.20.1 Free text input (Input)

```
string = 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 TRUE, only digits can be entered (no ”-” or ”.”, too!).

If *multiline?* is TRUE, 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.

Note the return value will be a string even if *numeric?* is TRUE.

See also 9.20.10 Set font for big messages (SetMessageFont)

9.20.2 Message (Message)

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

Shows the given text in a message window.

9.20.3 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.

See also 9.20.10 Set font for big messages (SetMessageFont)

9.20.4 Message with countdown/condition (SleepMessage)

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

See 9.10.2 Wait message with countdown / condition (SleepMessage) and 9.20.10 Set font for big messages (SetMessageFont)

9.20.5 Simple questions (Question)

```
int = 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.

Return values:
“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.

9.20.6 Selection from a list (Choice)

```
int = Choice( title, hint, default, timeout, value, value
              {, value} )
int = Choice( title, hint, default, 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.

See also 9.20.9 Set entry size and font for choices (SetChoiceEntryFormat)

9.20.7 Select directory (SelectDirectory)

```
string = SelectDirectory( title, message [, default] )
```

Shows a dialog to select an existing directory. If a *default* is given, the path is preselected if it exists.

9.20.8 Get filename (SelectFile)

string = **SelectFile(** *title, save?,* [*filter* [,*info* [,*default*]]] **)**

Shows a dialog to select a file.

If *save?* is TRUE, the user is able to enter a new filename, otherwise he can only select existing files.

The *filter* allows to show only files that fit to a file mask, like "*.txt" or "prefs.*".

The *info* works different for PC and the other versions. On the PC, the system's dialog is used. Since it doesn't allow additional texts (well, OK, it does, but way too complicated for a script language that should be fast and lightweight), the info text is shown in the file type description (where it usually says something like “All files (*.*)”). Since Windows Mobile's default dialog is quite terrible, an own dialog is used, which shows the info on top if one is given. If you want to use your script on both platforms, you might want to use a text that works for both, like “Select text file”.

If a *default* is given, the path/file is preselected (if *save?* is FALSE only if it exists).

9.20.9 Set entry size and font for choices (SetChoiceEntryFormat)

SetChoiceEntryFormat(*entry size* [, *font size, font name*] **)**

This modifies the height of each entry and – if given – the font of choice lists. That regards all choice dialogs shown after the command is invoked, including both the function and the control structure.

The *entry size* is in pixels. It will – like the font size – be doubled for mobile devices with VGA.

The *font size* is in points, the *font name* is e.g. “Courier”, “Tahoma”, etc. Keep in mind not all devices include all fonts, “Tahoma” (for mobile devices) or “Arial” (for desktop) usually are a good idea.

See also 8.4 Branching with selection dialog (Choice, ChoiceDefault) and 9.20.6 Selection from a list (Choice)

9.20.10 Set font for big messages (SetMessageFont)

SetMessageFont(*font size, font name* **)**

This modifies the font for message dialogs created by MortScript, i.e. BigMessage, SleepMessage, and Input. “Message” and “Question” use a system dialog which doesn't support own fonts.

The parameters are like the font parameters in SetChoiceEntryFormat above.

See also 9.20.3 Big message with scrollbar (BigMessage), 9.20.1 Free text input (Input) and 9.20.4 Message with countdown/condition (SleepMessage).

9.21 Status window

9.21.1 What is the status window?

The dialogs of 9.20 Dialogs interrupt the execution of the script, i.e. the script continues only after confirmation. So this isn't a nice way to tell the user what's going on along the way (e.g. for longer lasting operations or debugging).

For this reason, there's the status window. In this window, you can show messages without interrupting the script execution. There are two display styles for it: either only the last message is displayed (like BigMessage) or there's a list of the recent messages (similar to Choice dialog). You can add and remove messages even if the window's invisible.

The status window is only activated when it becomes visible or with special command (9.21.8 Show status window (StatusShow)). This way, the script can add and remove messages without disturbing the user with constantly popping up windows.

Optionally, the status window can contain a cancel button to abort the script and/or remain opened after the script finished (in this case, it has to be closed with an OK button).

9.21.2 Set display type (StatusType)

```
StatusType( style [, keep open? [, cancel button? ] ] )
```

The first parameter defines how the status window will look like. Possible values are:

- ST_HIDDEN – the status window will not be displayed
- ST_LIST – shows the messages in a list
- ST_MESSAGE – shows only the recent message

The existing messages will remain when the style is modified, e.g. when switching from ST_LIST to ST_MESSAGE the last message in the list will be displayed as single message.

If *keep open?* is TRUE, the window remains opened after the script was finished. It'll show an OK button which allows to dismiss the window.

If *cancel button?* is TRUE, a cancel button will be displayed during script execution. It allow to abort the script. This works similar to 9.22.9 End a running script (KillScript), but doesn't interrupt file operations.

9.21.3 Set window title and info text (StatusInfo)

```
StatusInfo( title [, info ] )
```

Allows to set the window title and a short info text which is displayed above the message resp. message list.

9.21.4 Set format for list entries (StatusListEntryFormat)

```
StatusListEntryFormat( entry height
                       [, font size, font name ] )
```

Defines the display style of messages in the list view. The parameters are the same as in 9.20.9 Set entry size and font for choices (SetChoiceEntryFormat).

9.21.5 Set number of elements in list (StatusHistorySize)

StatusHistorySize(*count* **)**

This sets the number of messages to show in the list view.

The given number of messages is also remembered if the status window is invisible or in single message view. This is handy for example, if you wish to show a protocol at the end of the script execution or write the messages to a file for debugging reasons (see 9.21.9 Write status messages to file (WriteStatusHistory)).

9.21.6 Add status message (StatusMessage, StatusMessageAppend)

StatusMessage(*message* [*style* [, *keep open?* [, *cancel?*]]] **)**
StatusMessageAppend(*text* **)**

StatusMessage adds a new message. In list view, it is appended to the end of the list and selected, so it will be visible. In single message view, only this new message will be displayed. The remaining parameters are the same as for 9.21.2 Set display type (StatusType). They allow to set the display style for the message in one go.

Please keep in mind the status window will not necessarily be visible (see 9.21.8 Show status window (StatusShow)). For warnings and similar messages, 9.20.2 Message (Message) and 9.20.4 Message with countdown/condition (SleepMessage) probably are the better solution.

With StatusMessageAppend, the text will be appended to the recent message. This is e.g. handy for “waiting dots” or success/fail states.

Example:
```
StatusMessage( "Step 1 ", ST_LIST, TRUE )
For i = 1 to 10
  StatusMessageAppend( "." )
Next
StatusMessageAppend( "OK" )
StatusMessage( "Finished" )
```

9.21.7 Delete status messages (StatusRemoveLastMessage, StatusClear)

StatusRemoveLastMessage()
StatusClear()

StatusRemoveLastMessage removes the recent message. This usually makes sense if the message should be “overwriteen”, e.g. changing “Copying files” to “Files were copied”. Until a new message is added, the previous message will be displayed resp. selected.

StatusClear removes all messages.

For both commands, you should consider that an empty status window (if no messages are available) doesn't look very good. So they should either be followed by a StatusMessage or used when the window is invisible.

9.21.8 Show status window (StatusShow)

`StatusShow()`

As already mentioned in 9.21.1 What is the status window?, the status window keeps in background if the user activated another application and new messages are added. With this command, the status window will become visible and active.

9.21.9 Write status messages to file (WriteStatusHistory)

`WriteStatusHistory(` *`file name`* `[,` *`append?`* `[,` *`code page`* `]]` **`)`**

This writes all stored messages (i.e. everything that is resp. would be displayed in list view) to a file.

The parameters are like in 9.13.2 Writing to a text file (WriteFile), only the contents comes from the messages of course.

9.22 Processes (running applications)

9.22.1 Process handling supported? (SupportsProcHandling)

bool = **SupportsProcHandling()**

Returns TRUE, if process functions like Kill and ProcExists are supported on the device.

This function is thought for PNAs, but is also supported on other devices.

The MortScript functions Kill, ProcExists, and ProcList require a system library (toolhelp.dll), which is not included on all “stripped down” Windows CE devices. Thus, if you try to use those functions on a device without it, it will cause an error message. With this function, you're able to avoid it (e.g. by using window functions as makeshift) or show your own error message.

9.22.2 Checking existence of a process (ProcExists)

bool = **ProcExists(** *process name* **)**

Returns TRUE, if the given process is running, FALSE if not.

The “process name” is the name of the EXE. It's usually better to give it without path, e.g. "solitaire.exe", because that's faster and less error prone (wrong path? typo? run from other location?). But it's also possible to check with the full path name. On desktop Windows, full path checks don't work for all programs, esp. for system threads and services it's not possible to query the path from the system.

9.22.3 Checking existence of a script process (ScriptProcExists)

bool = **ScriptProcExists(** *script name* **)**

Returns TRUE, if the given MortScript is running, FALSE 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 9.21.7 End a running script (KillScript).

9.22.4 List of running processes (ProcList)

Array = **ProcList(** [*with path?* [, *search string*]] **)**

Returns an array with currently running processes.

If *with path?* is TRUE, all elements in the array will contain the entire path name (e.g. “\Windows\sol.exe”), otherwise only the name of the program without path (e.g. “sol.exe”).

If a *search string* is given, only fitting processes will be returned. The comparison regards only the file name without path and is case insensitive. “test” would return “\path\to\Test.exe”, but not “\test\path\my.exe”.

9.22.5 List of running script processes (ActiveScripts)

Array = **ActiveScripts(** [*with path?* [, *search string*]] **)**

Similar to ProcList (see above), but returns the running MortScript processes. The elements contain the name of the script (e.g. “test.mscr”), not “MortScript.exe”.

9.22.6 Process name of active window (ActiveProcess)

string = **ActiveProcess(** [*full path?*] **)**

Returns the program name of the currently active window.

If “full path?” is TRUE, the complete path is included, otherwise it's the executable without path. With desktop Windows, it's not always possible to get the entire path.

9.22.7 Process name of given window (WindowProcess)

string = **WindowProcess(** window title [, full path?] **)**

Returns the program name of the window with the given title.

If “full path?” is TRUE, the complete path is included, otherwise it's the executable without path. With desktop Windows, it's not always possible to get the entire path.

9.22.8 End a running process (Kill)

Kill(*process name* **)**

Terminates the application. The parameter must be either the name of the exe without path (e.g. solitare.exe) or include the entire path. Like with ProcExists, you should prefer the version without path. See 9.21.2 Checking existence of a process (ProcExists) for more details.

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.).

9.22.9 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 9.18.4 Getting system paths (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
```

9.23 Signals

9.23.1 System volume (SetVolume, Volume)

```
SetVolume( value )
int = Volume()
```

Sets resp. retrieves the system volume. 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.

9.23.2 Play a WAV file (PlaySound)

```
PlaySound( WAV file )
```

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

9.23.3 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.

9.24 Display / screen

9.24.1 Get the color at a screen position (ColorAt)

```
int = 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.

9.24.2 Convert part of screen to characters (ScreenToChars)

```
Array = ScreenToChars( x, y, width, height, color
                       [, background color? [, char foreground
                       [, char background ] ] ] )
```

Converts a part of the screen to a text array, using a given color to separate between foreground and background.

For example, if there's a black circle on the screen, and ScreenToChars is invoked with color 0 (=black), it would be converted to an array like this:

```
array[1] = "___####___"
array[2] = "_########_"
array[3] = "##########"
array[4] = "##########"
array[5] = "##########"
array[6] = "_########_"
array[7] = "___####___"
```

So it's "only" something like an extended ColorAt, not a text recognition. It allows you to check more reliably than with single ColorAts to check whether a certain text or image is displayed. However, the result might become unusable if ClearType („softening" of font borders, a system option of windows) is enabled.

If *background color?* is set to TRUE, every color but the given one is treated as foreground. Otherwise, only the given color is treated as foreground.

By default, "#" is used for the foreground and "_" for the background. You can change this with the two *char* parameters.

9.24.3 Create the color code from RGB values (RGB)

```
int = 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.

9.24.4 Get the red/green/blue part of a color code (Red, Green, Blue)

```
int = Red( color )
int = Green( color )
int = Blue( color )
```

These functions are the counterpart to RGB. They extract the red, green, resp. blue part (0-255 each) of a color code returned by ColorAt or RGB.

9.24.5 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

9.24.6 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

9.24.7 Toggle display on/off (ToggleDisplay)

```
ToggleDisplay( on? )
```

Turns the display on (*on?* = TRUE) or off (*on?* = FALSE).

Not available for: PC, PNA, Smartphone

9.24.8 Get screen size (ScreenWidth, ScreenHeight)

```
int = ScreenWidth()
int = ScreenHeight()
```

Returns the width resp. the height of the screen in pixels.

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

`bool = Screen( type )`

Returns TRUE, if the screen fulfills the *type* condition, FALSE if it doesn't.
Allowed values for *type*:

"landscape"	(in landscape orientation?)
"portrait"	(in portrait orientation?)
"square"	(square screen?)
"vga"	(VGA resolution – no matter if “default” or “real/true VGA”)
"qvga"	(QVGA resolution)

Not available for: PC

9.24.10 Redraw today screen (RedrawToday)

`RedrawToday`

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

9.24.11 Show/hide wait cursor (ShowWaitCursor/HideWaitCursor)

`ShowWaitCursor`
`HideWaitCursor`

Shows (ShowWaitCursor) or hides (HideWaitCursor) the hourglass (or rotating disc in Windows Mobile). With desktop windows and on some PNAs, this might only work if a status window is displayed.

9.24.12 Get current mouse cursor (CurrentCursor)

`string = CurrentCursor( [Window] )`

Retrieves the type of the current mouse cursor of the given window. If no window name is given, it uses the current foreground window.

Possible return values are "arrow" (default arrow), "wait" (hourglass), "cross" (target cross), "help" (question mark), "uparrow", and "other" (e.g. application defined cursors).

This function does not necessarily return the cursor the user sees. For example, Windows Mobile shows the hourglass (or rather rotating disk) while applications are started or applications without window, desktop Windows might show an “application start” cursor (arrow with small hourglass) or window resize arrows. There's no reliable way to retrieve those cursors. It's only possible to get the cursor a window wants Windows to show.

9.24.13 Show/hide input panel (ShowInput/HideInput)

```
ShowInput
HideInput
```

Shows resp. hides the input panel (e.g. screen keyboard).

Not available for: PC, Smartphone

9.24.14 Set input panel type (SetInput)

```
SetInput( input type )
```

Sets the current input type, e.g. SetInput("Keyboard") or SetInput("Transcriber").
If you want to publish a script with this command, please regard the input types might be localized on other devices!

Not available for: PC, Smartphone

9.24.15 Determine clicked screen position (ScreenshotClick)

```
array = ScreenshotClick( [ message, x, y,
                          text color, back color ] )
```

Creates a screenshot of the current screen contents, shows it (usually, nothings seems to change, but the screen isn't updated anymore), and returns the clicked resp. tapped position. The first array element will contain the x position, the second the y position. If the window is closed otherwise (like a task manager or the OK button), “nothing” is returned (see 9.2.4 Check if variable is assigned (IsEmpty)).

The optional parameters allow to show a hint on the screenshot. X and y determine the upper left position of the message text, the colors must be given in the same format which is used by 9.24.1 Get the color at a screen position (ColorAt) and 9.24.3 Create the color code from RGB values (RGB).

This function is mostly meant for script developers, so they can easily determine the coordinates for ColorAt, MouseClick, and similar.

9.25 Clipboard

9.25.1 Copy text to the clipboard (SetClipText)

`SetClipText(` *`text`* **`)`**

Copies the given text to the clipboard.

9.25.2 Get text from the clipboard (ClipText)

`string` `=` **`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.

9.26 Memory

9.26.1 Available main memory (FreeMemory)

`int` `=` **`FreeMemory(`** `[`*`unit`*`]` **`)`**

Returns the available main memory. By default, it's in kilobytes, you can specify another unit by passing BYTES, KB, MB, or GB as parameter (constants, i.e. without quotes).

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("\")).

9.26.2 Size of the main memory (TotalMemory)

`int` `=` **`TotalMemory(`** `[`*`unit`*`]` **`)`**

Returns the total size of the main memory. By default, it's in kilobytes, you can specify another unit by passing BYTES, KB, MB, or GB as parameter (constants, i.e. without quotes).

9.27 Energy

9.27.1 Check if externally powered (ExternalPowered)

```
bool = ExternalPowered()
```

Returns TRUE, if the device is externally powered, FALSE if a battery is used.

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

9.27.2 Current battery level (BatteryPercentage, BackupBatteryPercentage)

```
int = BatteryPercentage()
int = BackupBatteryPercentage()
```

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

About the same goes for the backup battery. Be aware this isn't supported on every device, because some use condensers, some just don't support querying, and some don't have a backup battery (esp. cheap devices with not exchangeable battery or devices with WM5 and newer, which keep their data also without backup battery). It's device dependent what this function returns in those cases.

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

9.27.3 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

9.27.4 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

9.28 System

9.28.1 Get the system version (SystemVersion)

```
value = 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 (integer)
"minor"	returns the minor version number (integer)
"build"	returns the build number (integer)
"platform"	returns the platform, one of "Win95" (includes 98 and Me), "WinNT" (includes XP and Vista), and "WinCE" (includes Smartphone / PPC / Windows Mobile).

9.28.2 MortScript-Interpreter ermitteln (MortScriptExe)

```
string = MortScriptExe()
```

Returns the file name of the MortScript interpreters. It complies with SystemPath("MortScriptExe") \ "MortScript.exe". See also 9.18.4 Getting system paths (SystemPath).

9.28.3 Get the current MortScript variant (MortScriptType)

```
string = 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)

9.28.4 Get the current MortScript version (MortScriptVersion, GetMortScriptVersion)

```
string = MortScriptVersion()
GetMortScriptVersion( variable, variable, variable, variable )
```

Gets the version number of MortScript, either as string ("a.b.c.d") or into single variables (integer values).

When using the function MortScriptVersion, the parts are concatenated with dots, (e.g. "4.1.0.0"), the command GetMortScriptVersion assigns each part to a separate variable.

9.28.5 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.