MortScript V4.3

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

Inhaltsverzeichnis

1 Was ist MortScript? / Lizenz

MortScript ist "nur" ein Interpreter (so ähnlich wie die "Laufzeitumgebung" von Visual Basic), deshalb gibt es kein Programm, das für sich selbst irgendetwas sichtbares macht. (Außer die Registrierung der Dateierweiterungen, wenn MortScript.exe direkt aufgerufen wird, aber das ist nicht nötig, wenn du ein Installationsprogramm verwendet hast. Siehe auch 3 Installation)

Die Scriptsprache ist hauptsächlich auf die Automatisierung von Vorgängen ausgelegt ist, also andere Anwendungen starten und "fernsteuern" sowie grundlegende Systemoperationen wie Dateioperationen, Registry-Zugriffe usw. Deshalb stehen nur ein paar einfache Dialoge zur Verfügung.

Du musst heruntergeladene .mscr oder .mortrun Script-Dateien verwenden oder sie selbst mit irgendeinem Text-Editor schreiben (siehe 4.1 Scripte erstellen und ausführen). Für Anfänger kann es etwas kompliziert sein, sich an eigene Scripts heranzuwagen.

Du kannst diese Scripts im Datei Explorer ausführen wie jede andere Anwendung, also einfach antippen. Oder du erstellst im Startmenü (\Windows\Startmenü, ggf. auch anders übersetzt) eine Verknüpfung zu den Script-Dateien.

Ich übernehme keine Garantie für Schäden, die durch das Programm auftreten können (weder ich noch die Script-Autoren sind perfekt...). Beachte dabei auch, dass fremde Scripte auch recht gefährliche Möglichkeiten haben - so wie auch eine "normale" Anwendung Dateien lesen und löschen und Daten übers Internet verschicken kann.

MortScript ist Freeware, d.h., es kann ohne Bezahlung verwendet werden. Über eine kleine (oder große ;)) Spende als „Dankeschön“ und/oder „Entwicklungs-Anreiz“ würde ich mich aber trotzdem freuen. Siehe auch 11 Spenden.

Der Quelltext ist auf Anfrage verfügbar (per Subversion), im Gegensatz zur GPL darf jedoch kein eigenes Derivat erstellt werden. Damit soll vermieden werden, dass es zu „Du musst für das Script den XxxScript von Yyy verwenden!“-Verwirrungen kommt.

Die Auslieferung mit eigenen Scripts ist erlaubt, auch bei kommerziellen Scripts (wobei diese dann natürlich im Quelltext verfügbar sind...). Es muss aber an einer sinnvollen Stelle (readme.txt, Installationsprogramm o.ä.) darauf hingewiesen werden, dass MortScript ein Fremdprodukt mit evtl. anderer Lizenz ist, ein Link auf meine Web-Seite oder die Nennung meines Namens wäre nett (z.B. „MortScript ist Freeware, www.sto-helit.de“).

2 Funktionen

MortScript unterstützt u.a.:

- Starten, Aktivieren, Verstecken und Schließen von Programmen
- Wartefunktionen: Bestimmte Zeitspanne, warten auf Existenz oder Aktivierung von Fenstern.
- Senden von Tasten an Fester
- Senden von Mausklicks
- Kopieren, Umbenennen/Verschieben, Löschen, Links erstellen
- Anlegen und Entfernen von Verzeichnissen
- Unterstützung von ZIP-Archiven (kein Überschreiben von enthaltenen Dateien)
- Lesen und Schreiben von Textdateien
- Lesen und Schreiben in der Registry
- Internet: Lesen von Textdateien, Downloads, Verbindungsaufbau und -ende
- If-Bedinungen, Choice-Auswahlen und For-, ForEach-, While- oder Repeat-Schleifen
- Einige Systemfunktionen (z.B. Rotation, Lautstärke, Beleuchtung, Reset)
- Unterfunktionen, lokale Variablen, mehrstufige Arrays, ...

3 Installation

3.1 Verschiedene MortScript-Varianten

MortScript gibt es für PCs, PocketPCs, Smartphones (mit Windows Mobile) und PNAs (Navigationsgeräte auf Windows Mobile-Basis). Der Funktionsumfang variiert nach den Möglichkeiten der jeweiligen Geräte. Falls es eine Funktion für eine bestimmte Version nicht gibt, ist dies dort vermerkt. Man kann auch herausfinden, mit welcher Variante das Script ausgeführt wird (siehe 9.28.2 MortScript-Variante ermitteln (MortScriptType)).

In den Downloads befinden sich jeweils alle Varianten. Du musst dir die zu deinem System passende aussuchen. Dabei sind:

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

3.2 PC-Setup

Diese Installationsform gibt es nur unter Windows Mobile, also nicht für die PC-Version.

Einfach die MortScript-4.1-*System*.exe (z.B. MortScript-4.1-PPC.exe) aus dem exe-Verzeichnis des Archivs entpacken, auf dem PC ausführen und den Anweisungen folgen...

Derzeit gibt es kein Setup für die PC-Variante, siehe „Ausführbare Programme“ weiter unten.

3.3 CAB-Datei

Diese Installationsform gibt es nur unter Windows Mobile, also nicht für die PC-Version.

Kopiere die MortScript-4.1-*System*.cab aus dem cab-Verzeichnis des Archivs auf das Gerät (via „Durchsuchen“ in ActiveSync oder über eine Speicherkarte) und öffne sie dann über den Datei-Explorer auf dem Gerät (oder einer Alternative wie TotalCommander, Resco Explorer, u.ä.).

3.4 Ausführbare Dateien

Kopiere die enthaltenen Dateien aus dem nach dem Gerätetyp bezeichneten Unterverzeichnis des bin-Verzeichnisses im Archiv (z.B. „bin/PPC“) irgendwo auf das Gerät (z.B. nach "\Programme\MortScript") und starte dort MortScript.exe, damit die benötigten Registry-Einträge (für die Verknüpfungen zu den Script-Erweiterungen .mscr und .mortrun) angelegt werden.

4 Verwendung

4.1 Scripte erstellen und ausführen

MortScript führt Dateien mit der Erweiterung ".mscr" und ".mortrun" aus. Die letzteren um abwärtskompatibel zu sein, das Programm hieß früher "MortRunner".

Eine solche Datei kann mit jedem beliebigen Text-Editor erstellt werden. Zur Not geht auch PocketWord, mit diesem muss jedoch "Speichern als - Text" gewählt werden und die Erweiterung nachträglich von .txt in .mscr oder .mortrun umbenannt werden.

Wenn dein Editor mehrere Formate unterstützt, wähle bitte ANSI. Seit V4.1 werden auch Unicode-Dateien mit passendem Dateianfang unterstützt (siehe 9.13.1 Lesen einer Datei (ReadFile)), aus Kompatibilitätsgründen sollte aber dennoch soweit wie möglich ANSI verwendet werden.

Wenn die Datei – z.B. über den Datei-Explorer – geöffnet wird (einfach antippen), werden die Zeilen in dieser Datei werden nacheinander abgearbeitet - so wie in einer Batch-Datei.

Die erstellte Datei kann als Link ins Startmenü oder im Autostart-Verzeichnis aufgenommen werden. Mit dem „Datei-Explorer“ geschieht dies, indem die Datei „kopiert“ (PopUp-Menü bei Tippen&Halten auf dem Dateinamen) und in „\Windows\Start Menü“ oder „\Windows\AutoStart“ mit „Verknüpfung einfügen“ ein Link erstellt wird.

4.2 Parameter für MortScript.exe

Als Parameter für MortScript.exe muss das auszuführende Script mit dem kompletten Pfad angegeben werden. Sind Leerzeichen enthalten, muss diese Angabe in Anführungszeichen stehen.

Bei der PPC-Version gibt es zusätzlich den Parameter /wait=*n*, wobei *n* die Anzahl der Sekunden angibt, die MortScript auf das Existieren der angegebenen Datei wartet. Diese Funktion ist vorhanden, weil die Speicherkarten nicht sofort nach dem Einschalten zur Verfügung stehen. Wenn z.B. ein Script einer Anwendungstaste zugewiesen wurde, würde dies dazu führen, dass das Script auf der Speicherkarte nicht geöffnet werden kann. Standardmäßig wird 5 Sekunden gewartet.

Des weiteren werden alle Parameter im Format „name=wert“ als Variable für das Script definiert.

Alle anderen Parameter, die nicht mit „/“ oder „-“ beginnen, werden in der Reihenfolge des Erscheinens in den Array argv aufgenommen und in argc gezählt.

Die Variablen, die mit name=wert definiert werden, sind globale Variablen. Wenn lokale Variablen verwendet werden, müssen sie ggf. mit Global(...) ausgenommen werden. Siehe auch 7.5.2 Gültigkeitsbereich von Variablen.

Im Gegensatz dazu sind argv und argc nur in der Hauptroutine, also außerhalb von Sub, ansprechbar um Komplikationen mit Funktionsparametern (vgl. 8.11 Unterroutinen (Sub, Call/CallFunction, @...))

Beispiel:

```
pfad\zu\MortScript.exe "pfad\zu\script.mscr" opt1 meldung="Test" opt2
```

Definiert die globale Variable „meldung“ mit dem Wert „Test“, den Array „argv“ mit den Elementen argv[1] = „opt1“ und argv[2] = „opt2“, sowie den Parameterzähler argc mit dem Wert 2.

4.3 Mehrfaches Aufrufen und Abbrechen von Scripts

MortScript kann mehrmals laufen, aber jedes Script immer nur einmal.

Wenn ein bereits laufendes Script nochmal aufgerufen wird, werden Dialoge des laufenden Scripts (Choice, Message, ...) in den Vordergrund gebracht. Hat das laufende Script keine geöffneten Fenster, geschieht nichts.

Möchte man die laufenden Scripts beenden, können hierfür die Funktionen ScriptProcExists oder KillScript verwendet werden. Siehe auch die Informationen bei 9.22.9 Script beenden (KillScript).

5 Zusatz-Tools

5.1 Scripte beim Einstecken/Entfernen von Speicherkarten ausführen

Mit der Autorun.exe kann die Autostart-Funktion von Windows Mobile besser genutzt werden.

Beim Einstecken und Entfernen einer Speicherkarte für Windows das Programm „Autorun.exe“ im Ordner „2577“ (CE-Code für ARM-Prozessoren) oder „0“ aus (also z.B. „\Storage\2577\autorun.exe“).

Diese Funktion wird nicht auf allen Geräten unterstützt. Mir ist z.B. bekannt, dass sie HP beim iPAQ 2210 deaktiviert hat. Auf manchen Geräten muss der Ordner auch anders heißen.

Bei PNAs und PCs wird sie generell nicht unterstützt, die autorun.exe ist hier nur für „Dummy-Exen“ (s.u.) gedacht.

Wenn Autorun.exe, MortScript.exe sowie autorun.mscr und/oder autoexit.mscr in diesen Ordner kopiert werden, werden beim Einstecken autorun.mscr und beim Entfernen autoexit.mscr ausgeführt (sofern diese jeweils vorhanden sind).

Aus Kompatibilitätsgründen können auch autorun.mortrun und autoexit.mortrun verwendet werden. Wenn sowohl .mscr als .mortrun vorhanden sind, wird die .mscr-Datei ausgeführt.

5.2 „Dummy-Exe“ für Scripte

Wird die autorun.exe umbenannt, führt sie das passende Script aus. D.h., wird die autorun.exe z.B. in meinscript.exe umbenannt, wird meinscript.mscr oder, falls das nicht vorhanden ist, meinscript.mortrun ausgeführt.

Die umbenannte autorun.exe und das Script müssen dabei im selben Verzeichnis liegen.

Wenn im gleichen Verzeichnis auch eine MortScript.exe liegt, wird diese zum Ausführen des Scripts verwendet, ansonsten wird die verwendet, auf die die Verknüpfung zeigt. Dies setzt jedoch eine Installation voraus, sonst gibt es einen entsprechenden Fehler.

Dieses Feature ist v.a. für Programme sinnvoll, die andere Programme starten können, aber dort nur .exe-Dateien erlauben (also keine .mscr-Dateien), wie z.B. manche Profil-Tools für die PhoneEditions.

5.3 Hilfs-Scripts für Installationen (setup.dll)

Die setup.dll ist nur für PocketPCs verfügbar. Sie ermöglicht CAB-Dateien, Scripts automatisch nach einer Installation oder vor einer Deinstallation von CAB-Dateien auszuführen. Sie erspart es Entwicklern also in vielen Fällen, selbst eine Setup-DLL zu schreiben. Wenn CAB-Dateien für dich ein Buch mit sieben Siegeln sind, überspringe dieses Kapitel einfach... ;-)

Um die setup.dll zu aktivieren, muss sie als Setup-DLL angegeben werden. Beim CAB-Wizard von EVC geschieht dies mit `CESetupDLL = "setup.dll"` in der .inf-Datei, bei anderen Tools sollte es einen entsprechenden Menüpunkt oder eine Einstellung dafür geben.

MortScript.exe und – falls ZIP-Archive verwendet werden – mortzip.dll müssen vom CAB in das Standard-Anwendungsverzeichnis (%InstallDir%) installiert werden. Das gleiche gilt auch für install.mscr (wird nach der Installation ausgeführt) und uninstall.mscr (wird vor der Deinstallation ausgeführt), wobei diese aber auch weggelassen werden können. Was aber nur für jeweils eine davon sinnvoll ist, da die setup.dll ja sonst nichts bewirken würde...

6 Wichtige allgemeine Informationen

6.1 Begriffe

Konstante: Ein fester Wert, also Zahlen wie „100“ oder Zeichenfolgen wie "Test"

Variable: Eine Zeichenfolge die für einen zugewiesenen Wert steht.
Z.B. „x = 5“ (Variable x bekommt den Wert „5“) oder „Message(x)“ (Der x zugewiesene Wert „5“ wird ausgegeben).

Ausdruck: Eine Kombination aus Variablen, Konstanten, Funktionen (siehe unten) und Operatoren, die einen Wert ergibt (z.B. „5*x“ oder „"Script-Typ: " & ScriptType()“)

Zuweisung: Belegung einer Variable mit einem Wert, meist mit „*Variablenname* = *Ausdruck*“

Parameter: Ausdrücke, die Kommandos oder Funktionen übergeben werden.

Kommando: Eine Anweisung ohne Rückgabewerte, z.B. MouseClick

Funktion: Eine Anweisung, die einen Wert zurückliefert, z.B. SubStr. Wird in Ausdrücken verwendet, und kann nur in Zuweisungen oder Parametern verwendet werden.

Kontrollstruktur: Anweisungen, die den Ablauf des Programms beeinflussen, z.B. If, Choice, ...

6.2 Darstellung in dieser Anleitung

Der in dieser Anleitung verwendete Stil basiert locker auf der (E)BNF:

`fett`:	Fester Wert, z.B. das Kommando selbst
`kursiv`:	Variabler Wert, üblicherweise ein beliebiger Ausdruck
`[...]`:	Optional, kann weggelassen werden (meist werden dann Standardwerte verwendet)
`{...}`:	Kann wiederholt oder weggelassen werden
`x|y|z`:	Entweder x, y oder z muss angegeben werden (üblicherweise feste Werte).
`(...)`:	Gruppierung (üblicherweise um "|"-Optionen klarer darzustellen).

Wenn die Zeichen fett sind, müssen sie so angegeben werden, z.B. Klammern (**`(`**...**`)`**).

Generell gilt die folgende Syntax:
`Kommando` `[` **`(`** *`Ausdruck`* `{`**`,`** *`Ausdruck`* `}` **`)`** `]`
oder
`Variable` `=` *`Funktion`***`(`** `[` *`Ausdruck`* `{`**`,`** *`Ausdruck`* `}` `]` **`)`**
wobei ein allein stehender Funktionsaufruf nur eine besondere Form eines Ausdrucks ist.
Mehr dazu später unter 7 Mögliche Parameter bzw. Zuweisungen.

Bei (wenigen) Kommandos, die keine Parameter benötigen, sind die Klammern danach optional, d.h. es bleibt deinem Geschmack überlassen, ob du z.B. „RedrawToday“ oder „RedrawToday()“ schreibst. Dies gilt jedoch nicht für Funktionen! (Weil in Ausdrücken die Klammer bestimmt, ob das davor eine Variable oder ein Funktionsname ist.)

6.3 Leerzeichen, Tabulatoren und Zeilenumbrüche

Leerzeichen und Tabulatoren sind überall vor, nach und zwischen den einzelnen Elementen möglich. In Zeichenfolgen sind sie ein fester Teil derselben, ansonsten werden sie ignoriert.

Zeilenumbrüche innerhalb eines Befehls sind möglich, wenn am Ende der fortzusetzenden Zeile das Zeichen „\“ steht. Dies ist auch innerhalb von Zeichenfolgen möglich, es werden dann aber alle umgebenden Leerzeichen, Tabulatoren und der Zeilenumbruch selbst durch ein einzelnes Leerzeichen ersetzt.

Beispiel:

```
Message( "Das ist \
         ein Test" )
```

gibt den Text „Das ist ein Test“ aus.

Zeilenumbrüche in einer Zeichenfolge müssen mit ^NL^ angegeben werden, siehe auch 7.3 Feste Zeichenfolgen.

6.4 Groß- und Kleinschreibung

Bei Befehlen und Dateinamen wird die Groß-/Kleinschreibung nicht berücksichtigt, bei Fenstertiteln hingegen schon. Dafür sind aber auch Teile des Fenstertitels möglich.
D.h. `Show("WORD")` funktioniert nicht, aber `Show("Word")` aktiviert auch „Pocket Word“ (oder das erstbeste Fenster, das „Word“ im Titel enthält...)

6.5 Verzeichnisse und Dateien

Verzeichnisse und Dateien sollten immer mit absoluter Pfadangabe angegeben werden (z.B. `"\path\to\file.ext"` oder `"\some\directory"`), weil Windows Mobile kein "Aktuelles Verzeichnis" kennt – und auf dem PC könnte es nicht das sein, was du vermutest.

Wenn kein Pfad angegeben wurde, verwendet MortScript das Verzeichnis, in dem das gerade ausgeführte Script liegt – aber nur, wenn MortScript auf den Dateiinhalt zugreift, nicht für Systemoperationen. D.h., Include, CallScript(Function), ReadFile, WriteFile, ReadIni, WriteIni und ForEach mit IniSections oder IniKeys funktionieren auch mit relativen Pfaden, nicht aber Kommandos wie Copy, Move, Rename, alle ZIP-Operationen und so weiter.

Oder kurz gesagt: Alles was brauchbar ist um dein Script zu konfigurieren kann aus dem aktuellen Verzeichnis gelesen bzw. dorthin geschrieben werden, alles andere braucht vollständige Pfade.

Windows Mobile unterstützt kein „.“ und „..“ in Pfaden, nicht einmal in einer Form wie “\irgendein\pfad\..\datei.txt” (was z.B. das Ergebnis wäre, wenn man das Scriptverzeichnis davor hängt), d.h., man kann auf diese Art nicht auf Dateien eines übergeordneten Verzeichnisses zugreifen.

6.6 Kommentare

Kommentare sind möglich, indem das Zeichen # an den Anfang der Zeile gestellt wird. Leerzeichen vor dem „#“ sind erlaubt.

In INI-Dateien wird wie üblich ein Strichpunkt am Zeilenanfang als Kennzeichen für einen Kommentar verwendet.

7 Mögliche Parameter bzw. Zuweisungen

7.1 Ausdrücke

Alle Parameter für Funktionen und Kommandos, Bedingungen und mit „=“ zugewiesene Werte sind Ausdrücke. (Ausnahme: Alte Syntax, siehe Informationen am Ende der Anleitung)

Ausdrücke können aus den folgenden Bestandteilen bestehen:
Feste Zeichenfolgen in Anführungszeichen, z.B. `"Text"`
Feste numerische Werte, z.B. 42
Variablen, z.B. x
Funktionen, z.B. SubStr(*Parameter*)
Operatoren, z.B. +, -, &, ..., die bestimmen, wie Werte (Konstanten, Variableninhalte, Funktionsergebnisse) verbunden werden sollen.

Das klingt komplizierter als es ist. Ein paar Beispiele machen die Sache etwas klarer:

```
Message( "Hallo!" )
```
→ Das Kommando „Message“ wird mit der Zeichenfolge „Hallo!“ als Parameter aufgerufen

```
Sleep( 500 )
```
→ Das Kommando „Sleep“ wird mit der Zahl „500“ als Parameter aufgerufen

```
Sleep( pause * 100 )
```
→ Hier wird die Variable „pause“ mit dem Operator „*“ (Multiplikation) mit der Zahl „100“ verknüpft (in diesem Fall multipliziert).

```
Message( "Akkustand: " & BatteryPercentage() & "%" )
```
→ Verknüpft die beiden Zeichenfolgen mit dem Rückgabewert der Funktion „BatteryLevel()“ und übergibt diesen Wert an die Funktion „Message“.

```
meldung = "Akkustand: " & BatteryPercentage() & "%"
```
→ Wie oben, nur wird hier das Ergebnis der Variablen „meldung“ zugewiesen statt einem Kommando übergeben.

```
If ( BatteryPercentage() > 20 )
  # Befehle
EndIf
```
→ Ein Ausdruck als Bedingung

Wie die Konstanten und Variablen angegeben werden müssen und welche Operatoren es gibt, folgt in den nächsten Kapiteln.
Die möglichen Funktionen sind in 9 Kommandos und Funktionen aufgeführt.

7.2 Datentypen

MortScript kennt keine sog. „Typisierung“, Zahlen und Zeichenfolgen werden also bei Bedrarf ineinander umgewandelt.

Ob ein Wert als Zahl oder als Zeichenfolge interpretiert wird, kommt darauf an, in welchem Zusammenhang er verwendet wird.

Bei numerischen Operatoren (z.B. „+“, mehr dazu siehe Operatoren) werden die Werte ggf. zu Zahlen umgewandelt, „"5"+"10"“ würde also 15 zurückgeben. Wenn eine Zeichenfolge keine gültige Zahl enthält, wird „0“ (null) verwendet.

Umgekehrt wird bei Text-Operatoren (z.B. „&“, das Zeichenfolgen aneinander hängt) eine Zahl in eine Zeichenfolge umgewandelt, „5 & 10“ liefert also „510“ zurück.

Ähnlich verhält es sich bei Parametern. Hier ist meist aus dem Zusammenhang ersichtlich, welcher Datentyp erwartet wird. Ein auszugebender Text wird z.B. selbstverständlich als Zeichenfolge verwendet, eine Zeitspanne eine Zahl.

Bei Bedingungen und „An/Aus“-Parametern gilt folgende Regel: Entspricht der Wert einer gültigen Zahl außer 0, gilt dies als „Bedinung erfüllt“ bzw. „an“, andernfalls als „nicht erfüllt“ / „aus“.

D.h. Ausdrücke wie 5, "10", 1=1 u.ä. sind „wahr/an“, 0, "x", 2=1 sind „falsch/aus“.

Wenn eine Zeichenfolge einen Dezimalpunkt enthält, wird sie in eine Gleitkomma-Zahl umgewandelt und ggf., wenn eine ganze Zahl benötigt wird, gerundet. „SleepMessage("4.5", "Warte...", "Warten")“ würde z.B. 5 Sekunden warten. Wenn eine Gleitkomma-Zahl in eine Zeichenfolge umgewandelt wird, werden standardmäßig 6 Nachkommastellen verwendet. Du solltest die Format()-Funktion verwenden um bessere Ergebnisse zu bekommen.

Ein Sonderfall sind gefundene Fenster. Im Normalfall lassen sie sich wie Zeichenfolgen, die den Fenstertitel enthalten, verwenden. Zusätzlich wird aber auch ein sog. „Handle“ gespeichert. Mehr dazu unter 9.7.1 Fenstertitel und -variablen – Wie MortScript ein Fenster findet.

7.3 Feste Zeichenfolgen

Feste Zeichenfolgen müssen in Anführungszeichen (") eingeschlossen werden.
Um Anführungszeichen innerhalb von Anführungszeichen zu haben, müssen diese doppelt angegeben werden, also z.B.

```
Message( "Er sagte: ""Das ist ein Test""" )
```
→ Gibt den Text „Er sagte: "Das ist ein Test"“ aus.

Die folgenden Zeichenkombinationen werden mit dem entsprechenden Sonderzeichen ersetzt:

^CR^ → Wagenrücklauf (Carriage Return)
^LF^ → Zeilenvorschub (Line Feed)
^NL^ → Windows-/DOS-Zeilenumbruch in Dateien (New Line, besteht aus CR+LF)
^TAB^ → Tabulator

Unter Windows wird ein Zeilenumbruch üblicherweise mit der Kombination ^CR^^LF^ (= ^NL^) abgespeichert. Es gibt teilweise aber auch Dateien im Unix-Format, die nur ^LF^ enthalten.

7.4 Feste Zahlen

Zahlen können einfach als solche angegeben werden (also `x = 5`, `Sleep(20)`, ...).
Wenn Gleitkomma-Operationen erwünscht sind, muss ein Dezimalpunkt angegeben werden, z.B. „1.“ statt „1“.

7.5 Variablen

Variablen sind „Platzhalter“ für einen Wert, der ihnen zugewiesen wurde.

Als Variablen werden alle Bestandteile eines Ausdrucks interpretiert, die weder Konstante (z.B. 123 oder "Zeichenfolge"), Operator (+, -, &, ...) noch Funktionsaufruf (alles mit Klammern dahinter) sind.

Gültige Zeichen für Variablennamen sind die Buchstaben A-Z (keine Umlaute, Akzente o.ä.!), Ziffern und der Unterstrich („_“). Bei Variablennamen wird Groß- und Kleinschreibung nicht unterschieden, d.h. `MYVARIABLE` und `myvariable` bezeichnen denselben Wert.

Ein Variablenname darf nicht mit einer Ziffer anfangen, da er sonst in Ausdrücken für eine numerische Konstante gehalten würde (und alles danach für einen Operator oder etwas ungültiges). „9mod2“ ist also dasselbe wie „9 mod 2“, und nicht eine Variable!

Das Belegen von Variablen geschieht üblicherweise mit „=“, z.B.

myvar = 5 * x + y

Es gibt aber auch einige Kommandos und Kontrollstrukturen, die Variablen belegen, z.B. GetTime oder ForEach.

Zum Verwenden in Ausdrücken muss einfach nur der Variablenname angegeben werden, wie das „x“ im Beispiel oben.

Beachte ggf. auch, dass die Verwendung von Variablen in der alten Syntax etwas komplizierter war (%...% usw.).

7.5.1 Vordefinierte Variablen

Einige Variablen sind vordefiniert, um einige Ausdrücke etwas besser lesbar zu machen. Im Gegensatz zu anderen Sprachen könnte man sie ändern, dies ist aber nicht zu empfehlen:

`TRUE, ON, YES` sind mit 1 vorbelegt,

`FALSE, OFF, NO` sind mit 0 vorbelegt,

`CANCEL` ist mit 2 vorbelegt.

`PI` ist mit 3.14159265358979323846264338327 95 (π) vorbelegt

`SQRT2` ist mit 1.41421356237309504880168872420 97 (Quadratwurzel von 2) vorbelegt

`PHI` ist mit 1.61803398874989484820458683436 56 (φ, „goldener Schnitt“) vorbelegt

`EULER` ist mit 2.71828182845904523536028747135 27 (*e*) vorbelegt

7.5.2 Gültigkeitsbereich von Variablen

Normalerweise sind alle Variablen global, d.h., wenn du eine Variable belegst, kann ihr Wert auch in allen Sub-Blöcken (siehe 8.9 Unterroutinen (Sub, Call/CallFunction)) und Scripts, die mit CallScript (siehe 8.10 Anderes Script als Unterroutine (CallScript/CallScriptFunction)) aufgerufen wurden, abgefragt und modifiziert werden.

Wenn du lokale Variablen verwenden willst, musst du Local() oder Global() verwenden.

Wenn Local() ohne Parameter aufgerufen wird, werden alle Variablen, die danach verwendet werden, lokal verwendet bis der Sub-Block endet oder das Script beendet wird. Dies gilt auch für die Hauptroutine (den Code vor dem ersten Sub).

Übergibt man Local() Variablen als Parameter, werden nur diese Variablen lokal verwendet, während alle anderen global bleiben.

Global() arbeitet genau andersherum: Die dort angegebenen Variablen werden global verwendet, alle anderen lokal.

Beispiel:

```
Local()
x = "Test"
Call( "Sub1" )
Call( "Sub2" )
Message( x )     zeigt „Test“, weil die lokale Variable in Subs nicht geändert werden kann
Message( y )     zeigt nichts, weil es keine lokale Variable y gibt (nur die globale, s.u.)

Sub Sub1
  x = 5          setzt die globale Variable
  Local( x )
  y = "Hi!"      globale Variable! (nur x ist lokal!)
  Message( x )   zeigt nichts (die lokale Variable wurde nicht belegt!)
EndSub

Sub Sub2
  Global( x )
  Message( x )   zeigt den globalen Wert „5“
  Message( y )   zeigt nichts, weil es keine lokale Variable y gibt
EndSub
```

7.5.3 Arrays (Listen)

Arrays sind eine Sonderform der Variablen. Ein Array besteht aus mehreren zusammengehörigen Variablen, sogenannten Elementen.

Ein Element wird dabei durch den Variablennamen und den sogenannten „Index“ in eckigen Klammern dahinter angegeben, array[1] kennzeichnet also das Element „1“ des Arrays „array“.

Auch Zeichenfolgen sind als Index erlaubt, z.B. farben["blau"], wobei hier wie beim Variablennamen nicht zwischen Groß- und Kleinschreibung unterschieden wird (FARBEN["BLAU"] greift also auf den gleichen Wert zu).

Sowohl beim Zuweisen als auch bei der Verwendung können als Array-Index beliebige Ausdrücke verwendet werden. Dies ist der Hauptvorteil von Arrays, da der Zugriff auf die Elemente meist über eine Variable geschehen (z.B. eine Zählervariable).

Bei manchen Anweisungen (z.B. Choice oder Split) werden auch Array-Angaben unterstützt, dort werden aber nur die Elemente von 1 bis zur ersten nicht vorhandenen Zahl verwendet. Auf kleinere Indizes (<= 0), nach einer Lücke folgende Elemente oder Zeichenfolgen als Index werden dort also ignoriert.

Wenn numerische Indizes als Text übergeben werden, werden sie umgewandelt, sofern sie nicht mit einer 0 anfangen. Dies wurde gemacht, damit z.B. arr["007"] und arr["7"] auseinander gehalten werden können. Dagegen sprechen arr[7], arr[007] und arr["7"] dasselbe Element an. Ggf. ist es nötig mit Tricks wie arr[eingabe+0] zu arbeiten um einen numerischen Index zu erzwingen.

Weitere Ebenen können durch weitere Werte in eckigen Klammern verwendet werden, z.B. „Farben[x][y]“.

Beispiele:

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

liste[1] = "a"
liste[2] = "b"
liste["3"] = "c"
liste["0004"] = "d"
liste[5] = "f"
liste["a"] = "A"
idx = Choice( "Auswahl", "Wähle etwas", 0, 0, liste )
```

→ Hier werden nur „a“, „b“ und „c“ zur Auswahl angezeigt.

7.5.4 Referenzen ([Variablenname])

Referenzen erlauben es, auf eine Variable über einen Ausdruck zuzugreifen.

Man kann sie als eine Art Mischung aus „Alle Variablen sind Elemente eines unbenannten Arrays“ und Eval() (siehe 9.2.2 Ausdrücke in einer Zeichenfolge (Eval)) betrachten.

Um eine Variable zu referenzieren, benutze einfach einen Ausdruck, der einen gültigen Variablennamen (optional mit Array-Element) ergibt, in eckigen Klammern.

Zum Beispiel referenziert „["array[1]"]“ auf das erste Element von “array”. Natürlich ist das in dieser Form nicht sehr sinnvoll, weil „array[1]“ dasselbe bewirken würde und schneller zu parsen wäre. Aber mache daraus z.B. „[arrayName & "[" & elem & "]"]“, und die Vorteile werden klar.

Dies kann fast überall verwendet werden, einzige Ausnahme sind Kommandos in der alten Syntax ohne Klammern (siehe 10 Alte Syntax und Befehle).

Beispiele:

```
[zielVar] = [quellVar] * 10
Choice( "Auswahl", "Wähle:", [choiceArrayName] )
GetTime( [varStunde], [varMinute], [varSekunde] )
```

7.6 Operatoren

7.6.1 Auflistung der möglichen Operatoren

Alle möglichen Operatoren nach Priorität (höchste zuerst):

`()`	Klammern
`NOT`	Negation
`^`	Potenz (x^y → x^y)
`* / MOD`	Multiplikation, Division, Modulo (Rest von Divisionen)
`+ -`	Addition, Subtraktion
`& \`	Verknüpfung von Zeichenfolgen
`> >= < <= = <>`	Numerische Vergleiche
`gt ge lt le eq ne`	Alphanumerische Vergleiche
Bedingung `?` *Wahr* `:` *Falsch*	Liefert den „Wahr“-Wert, wenn die Bedingung erfüllt ist, sonst den „Falsch“-Wert
`AND &&`	Binäres / logisches Und
`OR ||`	Binäres / logisches Oder

7.6.2 Logische und binäre Operatoren

Bei logischen Operatoren (Wahr oder Falsch, also `&&`, `||` und `NOT`) gilt folgende Regel: Entspricht der Wert einer gültigen Zahl außer 0, gilt dies als „wahr“, andernfalls als „falsch“. Erfüllte Bedingungen liefern „1“ zurück.
D.h. Ausdrücke wie 5, "10", 1=1 u.ä. sind „wahr“, 0, "x", 2=1 sind „falsch“.
„NOT 5“ würde also „0“ zurückgeben, „NOT (2-2)“ ergibt „1“.

Der Unterschied zwischen `AND` und `&&` bzw. `OR` und `||` ist der, dass für `&&` und `||` jeder Wert, der als Zahl nicht 0 entspricht, wie 1 behandelt wird. Wenn du die Operatoren also nur für Vergleiche oder Funktionen, die etwas prüfen, verwendest, gibt es keinen Unterschied, da dort ohnehin nur die Werte 1 und 0 vorhanden sind.

Die binären Operatoren `AND` und `OR` sind zusätzlich noch für bitweise Abfragen hilfreich, so prüft z.B. „`( x AND 4 ) = 4`“ ab, ob im Wert der Variablen „x“ das 3. Bit gesetzt ist (4 = binär 100).
Die logischen Operatoren `&&` und `||` sind dagegen vor allem für „C-Hacker“ interessant, die gewohnt sind, dass 1 UND 2 nicht 0 (binär 01 AND 10 wäre 0) sondern 1 = „wahr“ ergibt.

7.6.3 Vergleiche

Numerische und alphanumerische Vergleiche haben dieselbe Priorität, sie wurden in der Liste oben nur der Übersicht wegen aufgeteilt.

Da es keine Typisierung gibt, muss es verschiedene Operatoren für numerische und alphanumerische Vergleiche geben. Dies bedeutet, dass `"123" < "20"` „falsch“ ist (weil 20 kleiner ist als 123), aber `123 lt 20` „wahr“ ist (weil in der alphabetischen Reihenfolge „1“ vor „2“ kommt, so wie „a“ vor „b“ kommt).

Wenn Du Dir die alphanumerischen Operatoren nicht merken kannst: Sie sind nur die Abkürzungen für „greater than“, „greater/equal“, „less than“, „(not) equals“, usw.

7.6.4 Verknüpfung von Zeichenfolgen und Pfaden

"\" ist ein Operator für die Verknüpfung von Pfaden. Es wird an der Verknüpfungsstelle genau einen "\" geben.
Im Gegensatz dazu hängt „&“ die einzelnen Werte einfach aneinander, wodurch ungültige Pfade entstehen können.

Beispiel:
```
"\My documents\" \ "\file.txt"
"\My documents" \ "file.txt"
"\My documents\" \ "file.txt"
```
→ liefern alle `"\My documents\file.txt"` zurück.

Dagegen:
```
"\My documents\" & "\file.txt"
```
→ "\My documents\\file.txt"
```
"\My documents" & "file.txt"
```
→ "\My documentsfile.txt"
```
"\My documents\" & "file.txt"
```
→ "\My documents\file.txt"

8 Ablaufstrukturen

8.1 Bedingungen

Als Bedingung kann jeder beliebige Ausdruck in Klammern verwendet werden. Die Bedingung ist erfüllt, wenn der zurückgelieferte Wert (ggf. nach einer Umwandlung zur Zahl) nicht 0 (Null) ist. Mögliche Funktionen sind bei der entsprechenden Gruppe unter „9 Kommandos und Funktionen“ aufgelistet. Lies auch das Kapitel 7, v.a. 7.2 Datentypen für weitere Informationen.

Beispiele:

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

While ( x <> 5 )
EndWhile
```

8.2 Einfache Verzweigung (If)

```
If( Ausdruck )
  { Anweisungen }
{ ElseIf( Ausdruck )
  { Anweisungen } }
[ Else
  { Anweisungen } ]
EndIf
```

Führt die Zeilen zwischen If und Else oder EndIf aus, wenn die Bedingung erfüllt ist, oder die Zeilen zwischen Else und EndIf (wenn es Else gibt), wenn sie es nicht ist.

Wenn ElseIf verwendet wird, werden nur die Zeilen zwischen der **ersten** erfüllten Bedingung und dem nächsten ElseIf, Else oder EndIf ausgeführt. Der Else-Block (wenn er vorhanden ist) wird nur ausgeführt, wenn **keine** Bedingung erfüllt wurde.

If, Else, ElseIf und EndIf müssen jeweils in einer eigenen Zeile stehen.

8.3 Verzweigung anhand von Werten (Switch)

```
Switch( Ausdruck )
Case( Wert {, Wert } )
  { Anweisungen }
{ Case( Wert {, Wert } )
  { Anweisungen } }
{ Default
  { Anweisungen } }
EndSwitch
```

Führt anhand des Wertes, den der angegebene Ausdruck hat, die Blöcke aus, die den Wert bei Case aufgelistet haben.

Die Anweisungen in Default-Blöcken werden ausgeführt, wenn kein vorheriger Case-Block ausgeführt wurde. Es sind mehrere Default-Blöcke möglich, wobei jeweils alle vorherigen Case-Blöcke berücksichtigt werden, also auch die vor einem vorherigen Default (natürlich innerhalb derselben Switch-Struktur). In der Praxis ist aber meist ein Default-Block am Ende die beste Wahl.

Die Werte können in mehreren Case-Blöcken auftauchen (z.B. „Case(1, 2)“ und „Case(2, 3)“). Die „passenden“ Blöcke werden dann der Reihe nach ausgeführt.

Ein „Durchrutschen“ wie in C ist in dieser Form nicht möglich. Durch mehrfache Angabe eines Wertes kann dies aber galanter und durchschaubarer erreicht werden.

Es können unterschiedliche Datentypen verwendet werden, das Ergebnis des Switch-Ausdrucks wird zum Vergleich in den Datentyp des jeweiligen Wertes umgewandelt.

Gleitkomma-Zahlen sind jedoch aus zwei Gründen etwas problematisch: Erstens kann es Rundungsfehler geben, zwei scheinbar gleiche Zahlen könnten sich z.B. in der 15. Nachkommastelle unterscheiden. Zweitens muss man aufpassen, dass nicht versehentlich ganze Zahlen verwendet werden. Case(2) würde z.B. dafür sorgen, dass das Ergebnis vom Switch-Ausdruck in eine ganze Zahl umgewandelt und somit gerundet würde. Damit würde der Block auch für Switch(1.5) oder Switch(2.4) erfüllt. Für einen Gleitkomma-Vergleich müsste Case(2**.**) verwendet werden.

8.4 Auswahldialog (Choice, ChoiceDefault)

```
( Choice( Titel, Hinweis, Wert, Wert {, Wert } )
| Choice( Titel, Hinweis, Array )
| ChoiceDefault( Titel, Hinweis, Standard, Zeit, Wert, Wert
                 {, Wert } )
| ChoiceDefault( Titel, Hinweis, Standard, Zeit, Array )
)
Case( Wert {, Wert } )
  { Anweisungen }
{ Case( Wert {, Wert } )
  { Anweisungen } }
{ Default
  { Anweisungen } }
EndChoice
```

Bringt eine Auswahlbox mit den angegebenen Werten. Bei Case muss der Index (1 für den 1. Eintrag, 2 für den 2., usw.) angegeben werten. Bei Cancel / keiner Auswahl wird 0 zurückgegeben (also „Case 0“).

Bei der Variante mit einem Array werden die Werte mit numerischen Indizes von 1 bis zum ersten unbelegten Element angezeigt, maximal jedoch 256.

Ansonsten ist die Funktionsweise identisch mit Switch.

Theoretisch könnte man also auch `Switch( Choice( ... ) )` (Choice als Funktion, siehe 9.20.6 Mehrfachauswahl (Choice)) verwenden, aber Choice als Ablaufstruktur ist übersichtlicher.

ChoiceDefault ist eine Variante, bei ein Eintrag vorausgewählt werden kann und der gerade gewählte Eintrag nach Ablauf der angegebenen Zeitspanne automatisch ausgewählt wird. Wählt der Benutzer einen anderen Eintrag, wird dieser Countdown neu gestartet.

Die Vorauswahl (*Standard*) muss als Index angegeben werden (z.B. 2 für den 2. Eintrag). Es ist auch 0 für keine Vorauswahl (also "Cancel", wenn der Benutzer nichts auswählt) möglich.

Die Zeitspanne für den Countdown muss in Sekunden angegeben werden. Wird 0 angegeben, gibt es keinen Countdown.

Siehe auch 9.20.9 Größe und Schriftart für Elemente in Auswahldialogen (SetChoiceEntryFormat)

Beispiel:

```
Choice( "Test","Wähle eine Zahl","Eins","Zwei","Drei" )
Case( 1 )
  Message( "Eins" )
Case( 2, 3 )
  Message( "Zwei oder Drei" )
Case( 3 )
  Message( "Drei" )
Case( 0 )
  Message( "Cancel" )
  Exit
EndChoice
```

8.5 Schleife mit Bedingung (While)

```
While( Bedingung )
     { Anweisungen }
EndWhile
```

Führt die Zeilen zwischen While und EndWhile aus solange die Bedingung erfüllt ist.
While und EndWhile müssen jeweils in einer eigenen Zeile stehen.

8.6 Schleife über mehrere Werte (ForEach)

```
ForEach Variable{, Variable } in Typ ( Parameter {, Parameter } )
  { Anweisungen }
EndForEach
```

Hierbei handelt es sich um ein recht mächtiges Instrument. Die angegebene(n) Variable(n) wird/werden in jedem Schleifendurchlauf mit den Werten gefüllt, die durch den gewählten Typ und Parameter vorgegeben werden.

Das reicht von einfachen Wertelisten (Typ "values") bis zu den Schlüsseln und Werten von Abschnitten in INI-Dateien ("iniKeys").

Bitte beachten: Wenn ein Array-Element als zu belegende Variable angegeben wird, wird der Index nur beim Betreten der Schleife ausgewertet. Das heißt, wenn „i“ beim Betreten der ForEach-Schleife „1“ ist und „array[i]“ als Variable angegeben wurde, wird bei jedem Durchlauf „array[1]“ belegt, auch wenn „i“ im Schleifenblock einen anderen Wert zugewiesen bekommt!

Dasselbe gilt für die Parameter: Sie werden ebenfalls ausgewertet, wenn die Schleife betreten wird.

Soweit nicht anders angegeben, können die Parameter als beliebige Ausdrücke angegeben werden (also auch Funktionsaufrufe, Operatoren, ...).

Derzeit gibt es hier die folgenden Möglichkeiten:

8.6.1 Schleife über Datenwerte (Liste von Ausdrücken, Array-Inhalt, geteilte Zeichenfolgen, Zeichen einer Zeichenfolge)

```
ForEach Variable in values ( Wert {, Wert } )
```

Weist der angegebenen Variablen die aufgelisteten Werte (jeweils beliebige Ausdrücke) nacheinander zu.

```
ForEach Variable in array ( Arrayvariable )
```

Weist der angegebenen Variablen die im Array enthaltenen Werte nacheinander zu. Dabei werden die Elemente von 1 bis zur ersten nicht vorhandenen Zahl verwendet, alphanumerische oder nach einer Lücke angegebene Indizes werden ignoriert.

ForEach *Index, Wert* **in array (** *Arrayvariable* **)**

Hierbei werden alle Werte des Arrays zurückgegeben. Der Index wird der ersten Variable und der zugewiesene Wert der zweiten Variable zugewiesen.

ForEach *Variable* **in split (** *Zeichenfolge, Trennzeichen, Kürzen?* **)**

Ähnlich der Split-Funktion (siehe 9.3.1 Aufteilen in mehrere Variablen/Arrayelemente (Split)), aber die einzelnen Teile werden der angegebenen Variablen nacheinander zugewiesen.

ForEach *Variable* **in charsOf (** *Zeichenfolge* **)**

Weist der Variablen jedes Zeichen der Zeichenfolge nacheinander zu. Dies funktioniert natürlich auch, wenn der Ausdruck eine Zahl ergibt – es werden dann z.B. „4“ und „2“ (für 42) nacheinander zugewiesen.

8.6.2 Schleife über Inhalte einer INI-Datei (Abschnitte, Werte)

ForEach *Variable* **in iniSections (** *Dateiname* [, *Kodierung*] **)**

Liefert die einzelnen Abschnitte („[Abschnitt]“, ohne eckige Klammern) der angegebenen INI-Datei.

Zur *Kodierung* siehe 9.13.1 Lesen einer Datei (ReadFile, ReadLine).

ForEach *Schüssel*, *Wert* **in iniKeys (** *Dateiname*, *Abschnitt*
[, *Kodierung*]**)**

Weist den angegebenen Variablen die Inhalte eines Abschnitts einer INI-Datei zu.

Schlüssel ist dabei die Variable, die den Namen vor dem „=“ erhält, *Wert* die Variable, die den Wert dahinter bekommt.

Zur *Kodierung* siehe 9.13.1 Lesen einer Datei (ReadFile, ReadLine).

8.6.3 Schleife über Registry-Inhalte (Unterschlüssel, Werte pro Schlüssel)

ForEach *variable* **in regSubkeys (** *Wurzel, Schlüssel* **)**

Liefert alle Unterschlüssel des angegebenen Schlüssels. Siehe 9.19.1 Registry-Einträge lesen (RegRead) für die Parameter.

ForEach *wert, inhalt* **in regValues (** *Wurzel, Schlüssel* **)**

Belegt die Variablen mit den Werten in einem Registry-Schlüssel.
„Wert“ ist der Name der Varialben, die den Namen des Werts zugewiesen bekommt, “Inhalt” wird der aktuelle Inhalt zugewiesen. Siehe auch 9.19.1 Registry-Einträge lesen (RegRead)
Diese Schleife liefert nicht den Standardwert eines Schlüssels (in Registry-Editoren üblicherweise als „(Default)“, „(Standard)“ oder „@“ angezeigt).

8.6.4 Schleife über Dateien und Verzeichnisse

```
ForEach Variable in files ( Such-Ausdruck )
ForEach Variable in directories ( Such-Ausdruck )
```

Liefert die Dateien bzw. Verzeichnisse zurück, die dem Such-Ausdruck entsprechen.
Der Ausdruck muss aus einem festen Pfad und einer Dateinamen-Angabe mit Platzhaltern bestehen, z.B. `"\My documents\A*.psw"`.

8.7 Feste Anzahl an Wiederholungen (Repeat)

```
Repeat ( Anzahl )
  { Anweisungen }
EndRepeat
```

Wiederholt die Befehle zwischen diesen beiden Befehlen *Anzahl* mal.
Die Anzahl muss mindestens 1 sein.

8.8 Einfaches Durchzählen (For)

```
For Variable = Start to Ende [ step Schrittweite ]
  { instructions }
Next
```

Im ersten Durchlauf wird *Variable* auf *Start* gesetzt, dann wird sie in jedem weiteren Durchlauf um *Schrittweite* erhöht (bzw. verringert, wenn *Schrittweite* negativ ist) bis *Ende* überschritten wurde (*Variable* ist größer als *Ende*, wenn *Schrittweite* positiv ist, kleiner, wenn *Schrittweite* negativ ist).

Wird die Schrittweite weggelassen, wird 1 oder -1 verwendet, je nachdem ob *Ende* größer als *Start* ist oder andersherum.

For arbeitet mit ganzen Zahlen und Gleitkommawerten. Bei Gleitkomma-Zahlen werden für den Vergleich 6 Nachkommastellen verwendet (siehe auch 9.4.8 Gleitkomma-Zahlen vergleichen (CompareFloat)).

Bitte berücksichtige, dass alle Ausdrücke (für *Start*, *Ende*, S*chrittweite*) nur einmal (beim Betreten der Schleife) ausgewertet werden. Wird also z.B. eine Variable für Ende oder Schrittweite verwendet („For i = 1 to end step x“), und diese in der Schleife verändert, werden für den nächsten Durchlauf und die Ende-Bedingung weiterhin die Werte verwendet, die vor der Schleife in den Variablen standen.

8.9 Fortsetzen und Abbrechen (Break, Continue)

```
Break[ ( Struktur-Typ ) ]
Continue[ ( Struktur-Typ ) ]
```

Mit Break wird eine Ablaufstruktur abgebrochen, d.h., das Script wird nach der entsprechenden End-Anweisung (EndWhile, Next, ...) fortgesetzt. Dies ist möglich für Switch, Choice, ChoiceDefault, While, ForEach, Repeat, For und Try.

Mit Continue wird der danach im selben Block folgende Code übersprungen und mit dem nächsten Schleifendurchlauf weiter gemacht. Gibt es keinen weiteren Durchlauf, wird wie bei Break nach dem Ende weiter gemacht. Dies ist möglich für alle Schleifen, also While, ForEach, Repeat und For. Bei Try ist Continue ebenfalls möglich, wird dort aber etwas anders gehandhabt, siehe 8.10

Normalerweise wird die innerste passende Ablaufstruktur abgebrochen bzw. fortgesetzt.

Beispiel:

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

Hier bewirkt das „Break“ gar nichts, da es sich auf die Switch-Anweisung bezieht und nur danach folgende Zeilen im Default-Block überspringen würde (selbstverständlich wäre es in einer If-Anweisung sinnvoller).

Das „Continue“ hingegen bezieht sich auf die While-Schleife, da es für Switch nicht möglich ist. Es würden also auch alle Anweisungen nach EndSwitch übersprungen und mit dem nächsten Durchlauf weiter gemacht.

Optional lässt sich angeben, auf welche Struktur sich Break oder Continue beziehen sollen. So ließe sich mit Break(BLOCK_WHILE) die gesamte While-Schleife beenden. Als Parameter sind möglich: BLOCK_FOR, BLOCK_FOREACH, BLOCK_REPEAT und BLOCK_TRY. Nur bei Break zusätzlich: BLOCK_SWITCH und BLOCK_CHOICE (auch für ChoiceDefault).

Es ist jedoch nicht möglich, sich auf eine äußere Struktur vom selben Typ zu beziehen, also z.B. bei einer While-Schleife in einer While-Schleife die äußere Schleife zu beenden.

8.10 Blöcke mit Fehlerbehandlung (Try, Catch)

```
Try
  Anweisungen
{ Catch
  Anweisungen }
EndTry
```

Hiermit wird die Fehlerbehandlung ein wenig vereinfacht.

Das Prinzip ist simpel: Wird innerhalb der Anweisungen nach Try „Break“ aufgerufen, werden sofort die Anweisungen im Catch-Block ausgeführt. Sinnvoll ist hier zumeist die Ausgabe bzw. das Loggen eines Fehlers und das Beenden des Programms oder das Zurückgeben eines Fehler-Flags (mehr zu Funktionen folgt später).

Mit „Continue“ innerhalb der Anweisungen nach Try wird direkt zur Anweisung nach EndTry gesprungen. Dies ist v.a. dann hilfreich, wenn man z.B. versucht, einen Wert aus mehreren

möglichen Quellen zu ermitteln und abbrechen will, sobald er gefunden wurde.

Beispiel:

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

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

  x = Input( "Text eingeben" )
  If ( x eq "" )
    Break
  EndIf
Catch
  Message( "x wurde weder gelesen noch eingegeben" )
EndTry
```

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

```
Sub Unterroutine
  { Anweisungen }
EndSub

Call( Unterroutine {, Parameter} )
CallFunction( Unterroutine, Variable {, Parameter} )

@Unterroutine( [ Parameter { , Parameter } ] )
value = @Unterroutine( [ Parameter { , Parameter } ] )
```

Mit „Call“, „CallFunction“ oder *@routine*(...) läuft das Script an der Zeile mit der „Sub“-Anweisung mit dem angegebenen Namen weiter.
Sobald das Ende der Unterroutine (EndSub oder ExitSub) erreicht wurde, wird an der aufrufenden Stelle weiter gemacht. Im Falle von *@routine*() als Funktionsaufruf kann das mitten in einem Ausdruck für einen Parameter der Fall sein.

Im Gegensatz zu den meisten anderen Anweisungen muss nach „Sub“ ohne Klammern der Name der Unterroutine stehen, Ausdrücke sind hier nicht erlaubt! Außerdem sind die „Parameter“ hier nicht die übergebenen Werte, sondern die Namen der Variablen, die diese Werte (die bei Call, CallFunction oder als @...()-Parameter übergeben werden) empfangen sollen.

Bei Call/CallFunction muss der Funktionsname in Anführungszeichen stehen, da er ein Ausdruck wie alle anderen Parameter ist. Theoretisch könnte man sogar etwas wie „Call(variableMitDerUnterfunktion)“ als eine Art „On ... Gosub ...“ (wenn sich noch jemand an den BASIC-Befehl erinnert...) verwenden, aber das ist kein guter Programmierstil und könnte Probleme verursachen, wenn dabei irgendeine Funktion herauskommt, die es nicht gibt.

Im Falle von „Call“ ist auch die alte Syntax eine nette Alternative, da „Call Unterroutine“ besser aussieht als „Call("Unterroutine")“. Das gilt aber nur, solange keine Parameter mitgegeben werden. Besser ist es allerdings, einfach „@Unterroutine()“ zu verwenden.

Wenn Parameter angegeben werden, werden in der Unterroutine zwei lokale Variablen (siehe 7.5.2 Gültigkeitsbereich von Variablen) belegt: argc enthält die Anzahl der übergebenen Parameter und argv ist ein Array mit allen Parametern. Wenn also zwei Parameter übergeben werden, ist argc 2, argv[1] der erste Parameter und argv[2] der zweite.

Wurden Parameter-Variablen bei „Sub“ angegeben, werden diese als lokale Variablen angelegt und mit den übergebenen Werten initialisiert. Wurden nicht genug Werte übergeben, bleiben die Variablen leer (IsEmpty()). Wurden mehr Werte übergeben, werden diese (und nur diese) in argc/argv abgelegt.

Wird CallFunction verwendet, wird ein Wert, der in der Unterroutine mit „Return(wert)“ oder „ExitSub(wert)“ gesetzt wurde, der angegebenen Variablen zugewiesen. Wird *@routine*(...) in einem Ausdruck verwendet, wird der Rückgabe so verwendet wie auch bei den internen Funktionen von MortScript (z.B. Input oder Length).

Beachte, dass im Gegensatz zu vielen anderen Sprachen Return() in MortScript die Unterfunktion nicht verlässt, sondern nur den Rückgabewert definiert! Siehe auch 8.14 Rückgabewert setzen (Return) und 8.15 Unterroutine verlassen (ExitSub).

Die Unterroutinen müssen nach dem Hauptprogramm kommen, MortScript beendet das Script beim ersten "Sub" auf das es stößt.

Beispiele:

```
@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( "Nicht genügend Parameter übergeben" )
    ExitSub( 0 )
  EndIf
  If ( argc > 0 )
    Message( "Mehr als zwei Parameter übergeben" )
  EndIf
  Return( p1 + p2 )
EndSub

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

8.12 Unterroutinen aus anderen Dateien einbinden (Include)

`Include(` *`Datei`* **`)`**

Mit Include werden alle Sub-Blöcke der angegebenen Datei so eingebunden als wären sie im aktuellen Script.

Die mit Include eingebundene Datei kann ebenfalls Include-Anweisungen enthalten, sollte aber sonst keine Kommandos außerhalb der Sub-Blöcke enthalten. (Diese würden einfach ignoriert)

Include-Anweisungen sollten am Anfang des Scripts stehen. Sie werden schon vor der Ausführung des Scripts interpretiert, da sonst evtl. Call-Anweisungen fehlschlagen würden.

Beachte bitte, dass mit vielen Includes auch die Gefahr steigt, dass zwei Sub-Blöcke mit demselben Namen auftauchen. MortScript bricht in diesem Fall schon vor der Ausführung des Scripts mit einer Fehlermeldung ab.

8.13 Anderes Script als Unterroutine (CallScript/CallScriptFunction)

CallScript(*MortScript-Datei* { , *Parameter* } **)**
CallScriptFunction(*MortScript-Datei, Variable* { , *Parameter* } **)**

Führt das angegebene Script aus als wäre es eine Unterroutine.

Dies gilt auch für die Variablen, d.h., es können auch die Variablen des aufrufenden Scripts verändert werden! Im aufgerufenen Script sollten also möglichst Local() oder Global() verwendet werden.

Auch die Parameterübergabe (argc und argv) und Rückgabe von Werten (Return()) arbeiten wie bei Sub-Unterroutinen.

Bitte beachte auch 6.5 Verzeichnisse und Dateien.

CallScript(Function) wurde als eine der ersten Möglichkeiten für so etwas wie Unterfunktionen eingeführt. Neue Scripts sollten besser Include und Call(Funktion) / @Funktion() verwenden.

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

Für den Aufruf von anderen Programmen oder „selbstständigen“ Scripten gibt es eigene Befehle, siehe 9.6 Ausführen von Anwendungen oder Dokumente öffnen.

8.14 Rückgabewert setzen (Return)

Return(*Wert* **)**

Gibt den angegebenen Wert zurück, wenn die Sub-Routine bzw. das Script mit Call(Script)Function aufgerufen wurde.

Im Gegensatz zu den meisten anderen Sprachen wird bei MortScript hiermit nicht die Unterfunktion bzw. das Script verlassen sondern nur der Rückgabewert gesetzt! Arrays sind hierbei auch möglich.

Siehe auch 8.9 Unterroutinen (Sub, Call/CallFunction) und 8.10 Anderes Script als Unterroutine (CallScript/CallScriptFunction)

8.15 Unterroutine verlassen (ExitSub)

ExitSub[**(** *Wert* **)**]

Verlässt die aktuelle Unterroutine sofort. Wird ein Wert angegeben, wird dieser wie bei Return zurückgeben. Andernfalls wird der mit Return gesetzte Wert oder „nichts“ (IsEmpty(...)) zurückgeben.

8.16 Script vorzeitig beenden (Exit)

Exit

Beendet das Script

9 Kommandos und Funktionen

Funktionen werden durch *Typ* **`= Funktion( ... )`** dargestellt.

Der Typ ist entweder *Zeichen(folge)*, *Bool(escher Wert)*, *Ganz(zahl)* oder *Gleit(kommazahl),* je nachdem, was die Funktion zurück gibt. Boolesche Werte sind eigentlich auch ganze Zahlen, aber es werden hier nur TRUE (1) oder FALSE (0) zurückgegeben.

Werden unterschiedliche Typen zurückgegeben, wird nur „*Wert* =“ angegeben. (Üblicherweise gibt es dann in der Beschreibung genauere Informationen.)

Selbstverständlich können alle Funktionen auch in komplexeren Ausdrücken als einer einfachen „Variable = ...“-Zuweisung verwendet werden (vgl. 7 Mögliche Parameter bzw. Zuweisungen).

9.1 Fehlerbehandlung (ErrorLevel)

ErrorLevel (*Fehlerebene* **)**

Bestimmt, welche Fehlermeldungen angezeigt werden. Die Fehlerebene muss eine Zeichenfolge (z.B. "syntax") sein, also **nicht** ErrorLevel(syntax) - außer „syntax“ wäre eine Variable, die "syntax" enthält...
Der Standard ist „error“.
Auch der Programmablauf wird ggf. beeinflusst: Wenn die Fehlerebene „warn“ oder „error“ ist, wird das Script bei Fehlern abgebrochen (siehe Liste unten).
Ist der ErrorLevel „off“ bis „syntax“, werden viele Fehler ignoriert, damit sie ggf. mit „wndExists(...)“ o.ä. abgefragt werden kann.

Mögliche Fehlerebenen:

off **Keine Fehlermeldungen**
Das Script wird ggf. ohne Nachricht abgebrochen.

critical **Kritische Fehlermeldungen**
derzeit keine, reserviert für spätere Versionen

syntax **Syntaxfehler**
z.B. falsche Parameteranzahl oder ungültige Bedingungen

error **Andere Fehler**
z.B. nicht existierende Fenster, Registry-Eintrag konnte nicht erstellt oder entfernt werden, neues Dokument oder Verzeichnis konnte nicht erstellt werden.

warn **Warnungen**
z.B. Datei/Verzeichnis konnte nicht entfernt werden, copy/move/rename schug fehl (Ziel bereits vorhanden?)

Die weiter unten stehenden Ebenen schließen die darüber jeweils mit ein, bei „error“ werden also auch Fehler der Stufe „syntax“ oder „critical“ ausgegeben.

9.2 Variablen

9.2.1 Variablen belegen („=“, „+=“, ... und Set)

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

Weist der Variablen das Ergebnis des Ausdrucks zu.

Set() sollte nicht mehr verwendet werden, verwende besser *Variable = Ausdruck*.

Die kombinierten Zuweisungsoperatoren (+= usw.) sind Abkürzungen für *Variable = Variable Operator Ausdruck*. `i+=1` bewirkt z.B. dassselbe wie `i=i+1`.

Allerdings hat Set eine kleine Extrafunktion: Wird als Variablenname eine Variable in %...% angegeben, wird die Variable belegt, die in dieser Variablen steht. Also wenn „varRef“ den Wert „var“ enthält, und %varRef% angegeben wird, wird nicht die Variable „varRef“ mit dem Ergebnis des Ausdrucks belegt, sondern „var“.

Enthält „varRef“ eine Zeichenfolge, die mit „%“ anfängt und aufhört, wird dieses Spielchen rekursiv weiter getrieben, bis ein Variablenname ohne umgebende % enthalten ist (oder das System wegen fehlendem Stack-Speicher die Grätsche macht...).

Dies ist üblicherweise recht verwirrend, fehleranfällig und etwas veraltet. Verwende besser Referenzen (siehe 7.5.4 Referenzen ([Variablenname])) wo immer es möglich und nötig ist.

9.2.2 Ausdrücke in einer Zeichenfolge (Eval)

```
Wert = Eval( Zeichenfolge )
```

Führt einen in der Zeichenfolge enthaltenen Ausdruck aus und liefert das Ergebnis zurück.

Beispiel:
```
x = Eval( "1+5*x" )
```
→ x = 26, wenn x vorher 5 war

9.2.3 Variable oder Array-Element entfernen (Clear)

```
Clear( Variable )
```

Entfernt den Inhalt der angegebenen Variablen oder des Array-Elements. Im Gegensatz zu einer leeren Zeichenfolge ("") liefert IsEmpty() TRUE (siehe unten) und ein gelöschtes Array-Element wird bei ForEach, Choice u.ä. nicht mehr berücksichtigt.

9.2.4 Prüfen ob eine Variable belegt ist (IsEmpty)

```
Bool = IsEmpty( Variable )
```

Liefert TRUE, wenn der Variable noch kein Wert zugewiesen wurde oder dieser mit Clear() entfernt wurde, FALSE wenn die Variable einen Wert enthält – selbst wenn es eine leere Zeichenfolge ("") ist.

9.2.5 Typ einer Variablen bestimmen (VarType)

```
Ganz = VarType( Variable )
```

Liefert den Typ einer Variablen.

Mögliche Werte:

- VAR_EMPTY (leer)
- VAR_INT (Ganzzahl)
- VAR_FLOAT (Gleitkommazahl)
- VAR_STRING (Text)
- VAR_ARRAY (Liste)
- VAR_WINDOW (Fenster)

9.2.6 Gültigkeitsbereich für Variablen (Local, Global)

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

Mit Local() werden alle (wenn keine Variablen angegeben wurde) oder die angegebenen Variablen lokal für den aktuellen Block (Sub-Routine oder die Hauptroutine) gehalten.

Global() arbeitet andersherum: Die angegebenen Variablen werden als globale Variablen behandelt, alle anderen sind lokal.

Siehe auch 7.5.2 Gültigkeitsbereich von Variablen.

9.3 Operationen mit Zeichenfolgen

9.3.1 Länge einer Zeichenfolge ermitteln (Length)

```
Ganz = Length( Zeichenfolge )
```

Liefert die Anzahl der Zeichen in der angegebenen Zeichenfolge.

Beispiel:
```
x = Length( "Dies ist ein Test" )
```
→ x = 17

9.3.2 Teil einer Zeichenfolge (SubStr)

```
Zeichenfolge = SubStr( Zeichenfolge, Start [, Länge ] )
```

Gibt „Länge“ Zeichen ab dem mit „Start“ angegebenen Zeichen zurück. Wird die Länge weggelassen, wird alles vom angegebenen Start bis zum Ende zurückgegeben.
Als „Start“ ist auch eine negative Zahl möglich. In diesem Fall wird vom Ende der Zeichenfolge ausgegangen. „-2“ kennzeichnet also das vorletzte Zeichen.

Beispiele:
```
x = SubStr( "abcdef", 2, 3 )
```
→ x = "bcd"

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

9.3.3 Einzelnes Zeichen einer Zeichenfolge (CharAt)

```
Zeichenfolge = CharAt( Zeichenfolge, Position )
```

Gibt das Zeichen an der angegebenen Position zurück. Ist die Zeichenfolge kürzer, wird „nichts“ zurückgeben (siehe 9.2.4 Prüfen ob eine Variable belegt ist (IsEmpty)).

9.3.4 Aufteilen und bestimmten Teil zurückgeben (Part)

```
Zeichenfolge = Part( Zeichenfolge, Trennzeichen, Index
                     [, Kürzen? ] )
```

Teilt die angegebene Zeichenfolge am angegebenen Trennzeichen (darf nur ein Zeichen sein!) und gibt den mit „Index“ angegebenen Teil zurück. Wird bei „Kürzen?“ ein Wert außer Null (z.B. TRUE) angegeben oder der Parameter weggelassen, werden umgebende Leerzeichen, Tabulatoren und Zeilenumbrüche entfernt, wird bei „Kürzen?“ Null (FALSE) angegeben, bleiben sie erhalten. Wird ein negativer Index angegeben, wird „von hinten“ gezählt, d.h. -1 liefert den letzten Teil, -2 den vorletzten, usw.

Beispiele:
```
x = Part( "a | b | c", "|", -1 )
```
→ x = "c" (letzter Teil, Leerzeichen wird entfernt)

```
x = Part( "a\ b \c.def", "\", 2, FALSE )
```
→ x = " b " (zweiter Teil, Leerzeichen bleiben erhalten)

```
x = Part( "eins, zwei, drei", ",", 4 )
```
→ x = "" (Leerstring bei nicht vorhandenen Teilen)

9.3.5 Zeichenfolge in einer Zeichenfolge finden (Find)

Ganz = **Find(** *Zu durchsuchende Zeichenfolge*, *Suchzeichenfolge* [, *Startposition*] **)**

Gibt die Position des ersten Zeichens zurück, ab dem die Suchzeichenfolge gefunden wurde.

Wurde eine Startposition angegeben, wird die Zeichenfolge erst ab diesem Zeichen durchsucht.

Ist die Suchzeichenfolge nicht enthalten, wird 0 zurückgegeben. Dies sollte abgefragt werden, um ungültige Anweisungen wie SubStr mit einer Startposition von 0 zu vermeiden.

Beispiele:
```
x = Find( "abcdefcd", "cd", 5 )
```
→ x = 7

```
x = Find( "abcdef", "CD" )
```
→ x = 0 (Groß-/Kleinschreibung wird berücksichtigt!)

9.3.6 Letztes Vorkommen eines Zeichens finden (ReverseFind)

Ganz = **ReverseFind(** *Zeichenfolge*, *zu suchendes Zeichen* **)**

Gibt die Position des letzten Vorkommens eines Zeichens in der Zeichenfolge zurück. Im Gegensatz zu Find ist hier nur ein einzelnes Zeichen erlaubt.

Ist das Zeichen nicht enthalten, wird 0 zurückgegeben.

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

9.3.7 Zeichenfolgen ersetzen (Replace)

Zeichenfolge = **Replace(** *Quelle*, *Alt*, *Neu* **)**

Ersetzt alle Vorkommen der *Alt*-Zeichenfolge in *Quelle* mit der *Neu*-Zeichenfolge

Beispiel:
```
x = Replace( "Meine alte Zeichenfolge", "alte", "neue" )
```
→ `x = "Meine neue Zeichenfolge"`

9.3.8 Zeichenfolge in Groß- oder Kleinbuchstaben umwandeln (ToUpper/ToLower)

```
x = ToUpper( Zeichenfolge )
x = ToLower( Zeichenfolge )
```

Gibt die angegebene Zeichenfolge in Groß- (ToUpper) bzw. Kleinbuchstaben (ToLower) zurück. Wird eine Variable übergeben, wird deren Inhalt nicht verändert.
Je nach System und Lokalisierung davon werden „Sonderzeichen“ wie Umlaute oder „è“ nicht berücksichtigt!

Beispiele:

```
x = ToUpper( "Abcba" )
```

→ x = "ABCBA"

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

→ x = "abcba"

9.3.9 Zeichen von/zu Unicode-Wert umwandeln (UcChar, UcValue)

```
Zeichenfolge = UcChar( Wert )
Ganz = UcValue( Zeichen )
```

UcChar liefert das Zeichen für einen bestimmten Unicode-Wert, UcValue liefert den Wert für ein bestimmtes Zeichen (einzelnes Zeichen als Parameter, ansonsten wird nur das erste Zeichen berücksichtigt!).

Beispiele:

```
x = UcValue( "A" )
```

→ x = 65

```
c = UcChar( x + 1 )
```

→ c = "B"

9.3.10 Teile eines Dateinamens (FilePath, FileBase, FileExt)

```
Zeichenfolge = FilePath( Dateiname mit Pfad )
Zeichenfolge = FileBase( Dateiname mit Pfad )
Zeichenfolge = FileExt( Dateiname mit Pfad )
```

Diese Funktionen helfen dabei, einen Dateinamen mit Pfad in seine Bestandteile zu zerlegen.

FilePath liefert den Pfad ohne Dateinamen („\My Documents“ für „\My Documents\test.txt“)

FileBase liefert den Dateiname ohne Pfad und Erweiterung („test“ für „\My Documents\test.txt“)

FileExt liefert die Erweiterung („txt“ für „\My Documents\test.txt“)

Kann v.a. zusammen mit 9.18.4 Systempfade ermitteln (SystemPath) praktisch sein.

9.4 Mathematische Funktionen

9.4.1 Formatierte Ausgabe (Format)

```
Zeichenfolge = Format( Wert, Nachkommastellen )
```

Liefert den Wert als Zeichenfolge mit der angegebenen Anzahl an Nachkommastellen. Der Wert wird ggf. kaufmännisch gerundet.

MortScript verwendet den Dezimalpunkt, damit Scripts international gleich funktionieren. Ggf. kann er mit Replace (9.3.6 Zeichenfolgen ersetzen (Replace)) durch das Komma ersetzt werden.

Beispiele:

```
x = Format( 123.456789, 2 )
```

→ x = "123.46"

```
x = Format( 12, 2 )
```

→ x = "12.00"

9.4.2 Konvertierung von/zu hexadezimalen Werten (NumberToHex, HexToNumber)

```
Zeichenfolge = NumberToHex( ganze Zahl )
Ganz = HexToNumber( Zeichenfolge )
```

NumberToHex konvertiert eine ganze Zahl (Gleitkommawerte werden gerundet) zu einer Zeichenfolge mit dem hexadezimalen Wert. Die Zeichenfolge wird so formatiert, dass immer ganze Bytes verwendet werden, also eine gerade Anzahl Zeichen (wie "0100" für 256 oder "0a" für 11).

HexToNumber arbeitet andersherum: Es konvertiert eine Zeichenfolge, die einen hexadezimalen Wert enthält, zu einer ganzen Zahl. Es arbeitet bis zum ersten ungültigen Zeichen, d.h. HexToNumber("axe") wird 10 zurückgeben, weil „a“ noch eine gültige hexadezimale Ziffer ist, „x“ jedoch nicht mehr. Großbuchstaben sind ebenfalls erlaubt ("ADE").

Bitte beachte, dass diese Funktionen nur mit Werten von 0 bis 2.147.483.647 (7fffffff) zuverlässig arbeiten. Zahlen von 2.147.483.648 (80000000) bis 4.294.967.295 (ffffffff) werden zwar problemlos von Dezimal in Hexadezimal umgewandelt, in die andere Richtung würden aber negative Zahlen zuückgeliefert. Außerdem liefern -1 bis -2.147.483.647 Hexadezimalwerte mit 8 Bytes von ffffffff (-1) bis 80000001 (-2.147.483.647). Das liegt daran, wie negative Werte intern abgelegt werden (das erste Bit ist das Kennzeichen für „negativ“).

Werte außerhalb dieses Bereichs verursachen Fehlermeldungen oder seltsame Ergebnisse.

9.4.3 Runden (Round, Floor, Ceil)

```
Ganz/Gleit = Round( Wert [, Nachkommastellen] )
Ganz/Gleit = Floor( Wert [, Nachkommastellen] )
Ganz/Gleit = Ceil( Wert [, Nachkommastellen] )
```

Liefert den gerundeten Wert. Werden Nachkommastellen angegeben, wird auf diese gerundet und die entsprechende Gleitzahl zurückgegeben (z.B. Round(2.56, 1) liefert 2.6), ansonsten wird die gerundete Ganzzahl zurückgegeben.

Floor rundet ab (2.9 → 2), Ceil rundet auf (2.1 → 3) und Round verwendet die kaufmännische Rundung, d.h. ab .5 wird aufgerundet (2.49 → 2,
2.5 → 3).

9.4.4 Zufallswerte (Rand)

```
Ganz = Rand( Max )
Gleit = Rand()
```

Wenn ein Maximalwert übergeben wird, wird eine ganze Zahl zwischen 0 und dem Maximalwert-1 zurückgegeben.

Wenn kein Parameter angegeben wird, werden Gleitkomma-Zahlen zwischen 0 und 0.999... zurückgeben.

9.4.5 Kreisfunktionen (Sin, Cos, Tan, etc.)

```
Gleit = Sin( Radiant )
Gleit = Cos( Radiant )
Gleit = Tan( Radiant )
Gleit = SinH( Radiant )
Gleit = CosH( Radiant )
Gleit = TanH( Radiant )
Gleit = ArcSin( Radiant )
Gleit = ArcCos( Radiant )
Gleit = ArcTan( Radiant )
```

ArcSin, -Cos und -Tan berechnen den Arkussinus, -cosinus bzw. -tangens; SinH, CosH und TanH den entsprechenden Hyberbolicus.

Bitte beachte, dass diese Funktionen den Radiant als Parameter benötigen! Wenn der Grad-Wert zur Verfügung steht (z.B. 45°), kann er mit “Grad * PI / 180” berechnet werden.

9.4.6 Logarithmen und Exponenten (Log, Log10, Exp)

```
Gleit = Log( Wert )
Gleit = Log10( Wert )
Gleit = Exp( Wert )
```

Log berechnet den Logarithmus auf der Basis von *e* (MortScript-Variable EULERT), Log10 auf der Basis von 10.

Exp(Wert) ist identisch mit EULERT ^ Wert.

9.4.7 Quadratwurzel (Sqrt)

```
Gleit = Sqrt( Wert )
```

Berechnet die Quadratwurzel des angegebenen Werts.

Die Quadratwurzel von 2 ist auch in der Variable SQRT2 abgelegt.

9.4.8 Gleitkomma-Zahlen vergleichen (CompareFloat)

```
Ganz = CompareFloat( Wert1, Wert2, Nachkommastellen )
```

Gleitkomma-Zahlen zu vergleichen ist etwas riskant, da es zu Rundungsfehlern kommen kann. Was wie eine Null aussieht, könnte gar keine „echte“ Null sein, sondern etwas wie $1*10^{-20}$.

Mit dieser kleinen Hilfsfunktion werden die Zahlen auf die angegebenen Nachkommastellen gerundet und dann verglichen.

Sie liefert -1 wenn Wert1 < Wert2 ist, 1 wenn Wert1 > Wert2 ist und 0 wenn beide Werte gleich sind.

9.4.9 Größten/kleinsten Wert ermitteln (Min/Max)

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

Liefert den größten (Max) bzw. kleinsten (Min) Wert der angegebenen Parameter zurück (mindestens 2 Parameter). Die Werte werden numerisch verglichen, der Rückgabewert entspricht dem des entsprechenden Parameters. D.h., Max(1, 2.5, "3.33") liefert die Zeichenfolge "3.33".

9.5 Arrays

9.5.1 Größten durchlaufenden numerischen Index ermitteln (MaxIndex)

Ganz = **MaxIndex(** *Array* **)**

Liefert den größten numerischen Index, der in einer durchlaufenden Reihe von 1 an definiert ist. Maximal jedoch 256.

Beispiel:

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

Hier wird 3 zurückgegeben.

MortScript konvertiert alphanumerische Indizes in Zahlen, wenn sie nicht mit 0 beginnen, so dass "2" auch berücksichtigt wird, während dies für "x" nicht gilt. Siehe auch 7.5.3 Arrays (Listen).

Weil 4 fehlt, wird 5 ignoriert. Es wäre genauso, wenn array[4] definiert und hinterher mit Clear() entfernt worden wäre.

9.5.2 Anzahl der Elemente ermitteln (ElementCount)

Ganz = **ElementCount(** *Array* **)**

Im Gegensatz zu MaxIndex wird hier die Anzahl aller Elemente, die je zugewiesen wurden, zurück gegeben. Dies gilt auch für alphanumerische Indizes und Elemente, deren Inhalt mit Clear() entfernt wurde.

Für den Array im Beispiel von MaxIndex würde 5 zurückgegeben.

9.5.3 Array aus einer Werteliste erstellen (Array)

Array = **Array(** *Wert* {, *Wert* } **)**

Liefert einen Array zurück, der die angegebenen Werte enthält. Die Indizes werden von 1 an durchnummeriert.

Bitte beachte, dass MortScript nicht ermöglicht, etwas wie „Array("a", "b", "c")[1]“ anzugeben, es kann nur eine Array-Variable belegt und später ausgelesen werden.

Beispiel:

```
days = Array( "So", "Mo", "Di", "Mi", "Do", "Fr", "Sa" )
day = days[ FormatTime("w")+1 ]
```

9.5.4 Array mit benannten Indizes erstellen (Map)

```
Array = Map( Index, Wert {, Index, Wert } )
```

Liefert einen Array mit den angegebenen Indizes und Werten.

Es muss eine gerade Anzahl an Parametern übergeben werden, wobei der erste Teil jedes Paares der Index und der zweite der Wert ist.

Bitte beachte, dass MortScript nicht ermöglicht, etwas wie „Map(1, "a", 2, "b")[1]“ anzugeben, es kann nur eine Array-Variable belegt und später ausgelesen werden.

Beispiel:

```
Monate = Map( "01", "Jan", "02", "Feb", "03", "Mär", etc. )
Monat = Monate[ FormatTime("m") ]
```

9.5.5 Aufteilen in mehrere Variablen/Arrayelemente (Split)

```
Split( Zeichenfolge, Trennzeichen, Kürzen?, Variable
     {, Variable } )
Array = Split( Zeichenfolge, Trennzeichen [, Kürzen? ] )
```

Teilt die Zeichenfolge an den Trennzeichen. Wenn diese Angabe aus mehreren Zeichen besteht, müssen sie so nacheinander in der aufzuteilenden Zeichenfolge vorkommen.

Wird die Kommando-Version mit mehreren Variablen verwendet, werden die Teile den angegebenen Variablen zugewiesen. Überflüssige Variablen bleiben/werden leer, Teile für die keine Variable angegeben wurde, werden ignoriert.

Wird nur eine Variable angegeben oder die Funktion verwendet, erhält die angegebene Variable einen Array mit den einzelnen Teilen, also *Variable*[1] bis *Variable*[*n*]. Jedes Element ist eine Zeichenfolge.

Ist "Kürzen?" TRUE (oder bei der Funktion weggelassen), werden die Teile umgebende Leerzeichen, Tabulatoren und Zeilenumbrüche entfernt.

Beispiele:

```
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( "eins, zwei, drei", ",", 1, liste )
```
→ liste[1]="eins", liste[2]="zwei", liste[3]="drei"

```
liste = Split( "eins, zwei, drei", "," )
```
→ liste[1]="eins", liste[2]="zwei", liste[3]="drei"

9.5.6 Array-Elemente zu Zeichenfolge verbinden (Join)

```
Zeichen = Join( Array [, Verbindung ] )
```

Join ist der Gegenpart zu „Split“. Es verbindet alle Elemente mit numerischen Indizes von 1 bis zum ersten unbelegten Index zu einer Zeichenfolge.

Wenn *Verbindung* angegeben wurde, wird diese Zeichenfolge jeweils zwischen die Elemente eingefügt.

Beispiele:

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

→ str = "a|b|c"

```
arr[1] = "Dies"
arr[2] = "ist"
arr["3"] = "ein"
arr[04] = "Test"
arr["test"] = "von"
arr[6] = "Join"
str = Join( arr )
```

→ str = "DiesisteinTest" ("3" wird in einen numerischen Index umgewandelt, "test" ist alphanumerisch, 5 fehlt, weswegen 6 nicht berücksichtigt wird)

9.5.7 Array auf enthaltenes Element prüfen

```
Bool = InArray( Array [, Element ] )
```

Überprüft, ob das angegebene *Element* im angegebenen *Array* enthalten ist. Es werden hierbei die Werte geprüft, Indizes lassen sich ggf. mit IsEmpty(array[index]) prüfen.

Der Vergleich geschieht ähnlich wie bei Switch (8.3), der Datentyp des Elements bestimmt also die Art des Vergleichs.

9.6 Ausführen von Anwendungen oder Dokumente öffnen

9.6.1 Anwendung/Dokument öffnen und Script fortsetzen (Run)

Run(*Applikation* [, *Parameter*] **)**

Startet die Anwendung. Es gehen auch Links (*.lnk), Parameter und Dokumente.
Der Pfad muss vollständig angegeben sein.

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

9.6.2 Anwendung/Dokument öffnen und warten (RunWait)

RunWait(*Applikation* [, *Parameter*] **)**

Wie Run, aber hier wird auf das Beenden des Programms gewartet.
Hier sind leider keine .lnk-Angaben möglich.

Beachte, dass dieser Befehl nicht die gewünschte Wirkung hat, wenn das Programm bereits läuft. In diesem Fall startet Windows dieses Programm nämlich ein zweites Mal. Diese zweite Instanz sucht dann nach einer bereits laufenden Instanz. Wird sie gefunden, wird das bereits laufende Programm aktiviert und die zweite Instanz beendet.
D.h., das Script läuft nach dem Beenden der zweiten Instanz sofort weiter, es wird nicht auf das Ende der bereits laufenden Instanz gewartet!

9.6.3 Anderes Script als Unterroutine (CallScript)

CallScript(*MortScript-Datei* {, *Parameter* } **)**
CallScriptFunction(*MortScript-Datei, Variable* {, *Parameter* } **)**

Ruft ein anderes Script auf, als wäre es eine Unterroutine.

Siehe 8.10 Anderes Script als Unterroutine (CallScript/CallScriptFunction)

9.6.4 Neues Dokument/Element erstellen (New)

New(*Menüeintrag* **)**

Erstellt ein neues Dokument (bzw. Termin o.ä.).
Der Menüeintrag muss genau so angegeben werden, wie er im Menü "Neu" im Heute-Bildschirm steht. Achte darauf, dass dieser Befehl von der Lokalisierung abhängig ist!

Leider ist diese nützliche Funktion seit Windows Mobile 5 nicht mehr so leicht zu erreichen. Hier muss entweder auf gut Glück ausprobiert werden, oder in der Registry unter HKEY_LOCAL_MACHINE\Software\Microsoft\Shell\Extensions\NewMenu geschaut werden.

Nicht verfügbar auf: PC

Beispiel:
```
New(  "Termin" )
```

9.6.5 Anwendung zu bestimmtem Zeitpunkt ausführen (RunAt)

```
RunAt( Unix-Zeitstempel, Applikation [, Parameter] )
RunAt( Jahr, Monat, Tag, Stunde, Minute
     , Applikation [, Parameter ] )
```

Startet das angegebene Programm zum angegebenen Zeitpunkt. Dazu wird das Programm in die sog. „Notification Queue“ eingetragen. Der PPC schaltet sich wenn nötig automatisch zum angegebenen Zeitpunkt ein.

Der „Unix-Zeitstempel“ ist die Zeit in Sekunden seit dem 01.01.1970. Diese Variante ist v.a. in Kombination mit TimeStamp() interessant, z.B. TimeStamp()+86400 für eine Ausführung in 24 Stunden (* 60 Minuten * 60 Sekunden = 86400).

Auf vielen Geräten können MortScripts nicht direkt ausgeführt werden. Statt dessen muss MortScript.exe mit dem Script als Parameter aufgerufen werden, z.B.

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

Ein weiteres Problem: Auf vielen PPCs mit WM5 wird das Gerät zwar eingeschaltet und das Programm gestartet, aber das Display bleibt aus und das Gerät schaltet sich kurz nach dem Start des Programms wieder ab. Teilweise hilft ein ToggleDisplay(ON) am Anfang des aufgerufenen Scripts, wo nicht, helfen leider nur System-Updates und z.T. Registry-Hacks.

Nicht verfügbar auf: PC

9.6.6 Anwendung beim Einschalten ausführen (RunOnPowerOn)

```
RunOnPowerOn( Anwendung [, Parameter ] )
```

Führt ein Programm automatisch bei jedem Einschalten des Geräts aus. Dazu wird das Programm in die sog. „Notification Queue“ eingetragen.
Der Einsatz sollte gut überlegt werden, da es z.B. zu lästigen Fehlermeldungen führen kann, wenn das angegebene Programm gelöscht oder verschoben wird.

Bitte die Hinweise bei RunAt bzgl. Aufruf von Scripten und WM5 beachten!

Nicht verfügbar auf: PC

9.6.7 Anwendung aus der „Notification Queue“ entfernen

```
RemoveNotifications( Anwendung [, Parameter] )
```

Entfernt das angegebene Programm aus der „Notification Queue“, d.h., es wird nicht mehr automatisch zu bestimmten Zeitpunkten (RunAt) oder Ereignissen (RunOnPowerOn) ausgeführt. Gibt es mehrere Einträge (z.B. mehrere „RunAt“s) werden alle entfernt.

Wird ein Parameter angegeben, wird dieser überprüft und nur die Einträge mit dem entsprechenden Parameter entfernt. Andernfalls werden alle Aufrufe des angegebenen Programms entfernt, egal welcher Parameter in der Notification eingetragen ist. Um nur die Aufrufe ohne Parameter zu entfernen, muss eine leere Zeichenfolge ("") übergeben werden.

Nicht verfügbar auf: PC

9.7 Fenster

9.7.1 Fenstertitel und -variablen – Wie MortScript ein Fenster findet

Viele Befehle und Funktionen in diesem Abschnitt (und auch einige andere, z.B. diverse Send...-Befehle) benötigen eine Fenster-Angabe.

Wird ein Fenstertitel als Zeichenfolge übergeben, sucht MortScript dann nach einem Fenster, das den angegebenen Text im Fenstertitel enthält. Dabei wird Groß- und Kleinschreibung unterschieden, d.h. „Word“ findet kein Fenster, das „WORD“ enthält. Gibt es mehrere passende Fenster, versucht MortScript anhand der folgenden Kritieren (das oberste ist das wichtigste) das „Bestmögliche“ herauszufinden:

- Ist das Fenster ein Hauptfenster? (kein Elternfenster, oder das Elternfenster ist der Desktop)
- Wurde der angegebene Text am Anfang des Fenstertitels gefunden?
- Ist das Fenster sichtbar? (dies hat nichts damit zu tun, ob das Fenster im Hintergrund ist, eher damit, ob es in der Taskliste aufgelistet wird).

Als Alternative gibt es die Möglichkeit, statt dem Titel einen „Fenster-Wert“ zu übergeben.

Dieser spezielle Typ wird von ActiveWindow(), FindWindow() und FindWindows() zurückgegeben.

Wird solch eine Variable (oder auch der Rückgabewert direkt) angegeben wo eine Zeichenfolge erwartet wird, wird der Fenstertitel verwendet (etwa in Message(ActiveWindow())). Zusätzlich wird aber intern auch ein so genanntes „Window handle“ abgelegt, das es MortScript ermöglicht, das Fenster schneller und ohne Verwechslungsgefahr anzusprechen.

Bei Operationen geht diese Zusatzinformation ggf. verloren (z.B. bei fenster = fenster & " Dokument1").

9.7.2 Fenster in den Vordergrund bringen (Show)

Show(*Fenster* **)**

Aktiviert das Fenster mit dem enstprechenden Titel.

9.7.3 Fenster in den Hintergrund verschieben (Minimize)

Minimize(*Fenster* **)**

Versteckt (minimiert) das Fenster mit dem enstprechenden Titel.

9.7.4 Fenster schließen / Anwendung beenden (Close)

Close(*Fenster*)

Schließt (beendet) das Fenster mit dem enstprechenden Titel. Wenn es sich dabei um das Hauptfenster eines Programms handelt, wird es dadurch üblicherweise beendet. Einige wenige Programme ignorieren dies jedoch.

9.7.5 Aktives Fensters ermitteln (ActiveWindow)

```
Fenster = ActiveWindow()
```

Gibt das gerade aktive Fenster zurück (Titel und Handle, siehe 9.7.1 Fenstertitel und -variablen – Wie MortScript ein Fenster findet).

9.7.6 Prüfen, ob ein Fenster aktiv ist (WndActive)

```
Bool = WndActive( Fenster )
```

Gibt TRUE zurück, wenn das angegebene Fenster aktiv ist, FALSE wenn nicht.

9.7.7 Fenster suchen (FindWindow, FindWindows)

```
Fenster = FindWindow( Titel [, Position [, Groß-/Kleinschr.?
                    [, nur Anwendungen? [, unsichtbare?
                    [, Klasse ]]]]] )

Array = FindWindows( [ Titel [, Position [, Groß-/Kleinschr.?
                    [, nur Anwendungen? [, unsichtbare?
                    [, Klasse ]]]]] )
```

Sucht nach dem Fenster bzw. den Fenstern mit den angegebenen Eigenschaften:

- *Titel*: Der Fenstertitel bzw. ein Teil davon
- *Position*: Eine der vordefinierten Konstanten (Standard: ANYWHERE):
  - ➔ ANYWHERE = *Titel* taucht irgendwo im Fenstertitel auf
  - ➔ BEGINNING = *Titel* taucht am Anfang des Fenstertitels auf
  - ➔ COMPLETE = *Titel* enthält den kompletten Fenstertitel
- *Groß-/Kleinschr.?*: Soll die Groß-/Kleinschreibung beim Vergleich berücksichtigt werden? (Standard: nein)
- *nur Anwendungen?*: Berücksichtigt nur Fenster, die bestimmte für „Hauptfenster“ übliche Attribute haben, z.B. kein Elternfenster (oder den Desktop). Dadurch werden z.B. Dialogelemente, die für Windows ebenfalls eigene Fenster sind, weggelassen. (Standard: nein)
- *unsichtbare?*: Sollen auch unsichtbare Fenster aufgelistet werden? (Standard: nein)
- *Klasse*: Hier kann eine Fensterklasse angegeben werden. Der Vergleich ist hier immer vollständig und mit Groß-/Kleinschreibung.

FindWindow liefert nur ein einzelnes Fenster zurück. Falls es mehrere passende gibt, wird nach einer internen Gewichtung ausgewählt. Z.B. zählt ein Fenster, bei dem der angegebene Titel am Anfang auftaucht höher als eines, bei dem der Text in der Mitte auftaucht und ein Anwendungsfenster mehr als ein Unter-Fenster. Der Fenstertitel muss hier angegeben werden, da sonst zu viele Fenster in Frage kommen. Wird kein passendes Fenster gefunden, wird ein leerer Wert (siehe 9.2.4 Prüfen ob eine Variable belegt ist (IsEmpty)) zurück gegeben.

FindWindows liefert alle passenden Fenster als Array zurück. Dies gilt auch, wenn nur ein Fenster (nur ein Element) oder kein Fenster (leerer Array) gefunden werden.

9.7.8 Prüfen, ob ein Fenster existiert (WndExists)

```
Bool = WndExists( Fenster )
```

Ähnlich wie WndActive(), liefert aber auch dann 1, wenn das Fenster irgendwo im Hintergrund existiert.

9.7.9 Warten auf Existenz eines Fensters (WaitFor)

```
WaitFor( Fenster, Sekunden )
```

Wartet (max. die angegebene Zeit) bis das angegebene Fenster existiert.

9.7.10 Warten auf Aktivierung eines Fensters (WaitForActive)

```
WaitForActive( Fenster, Sekunden )
```

Wartet (max. die angegebene Zeit) bis das angegebene Fenster aktiv ist.

9.7.11 Fenstertitel / Elementinhalt ermitteln (WindowText)

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

Ermittelt die Beschriftung des Elements, das sich an der angegebenen Stelle befindet.

Dies funktioniert v.a. bei Eingabefeldern, Beschriftungen und Buttons von Standarddialogen recht gut. Problematisch sind dagegen vom Programm "gemalte" Dialoge (z.B. in Spielen) oder Listen.

In diesen Fällen wird meist der Programmtitel oder nichts (Leerstring) zurückgegeben.

9.7.12 Fensterklasse ermitteln (WindowClass)

```
Zeichen = WndClass( Fenster )
```

Ermittelt die Klasse des angegebenen Fensters. Es handelt sich hierbei um eine interne Bezeichnung, die von Windows oder der Anwendung vergeben wird, z.B. „Dialog“, „Edit“, etc.

9.7.13 Fensterposition ermitteln (GetWindowPos, WndLeft, -Right, -Top, -Bottom)

```
GetWindowPos( Fenster, links, oben, rechts, unten )
Ganz = WndLeft( Fenster )
Ganz = WndRight( Fenster )
Ganz = WndTop( Fenster )
Ganz = WndBottom( Fenster )
```

Liefert die Position des angegebenen Fensters.

GetWindowPos setzt die Variablen, die für links, oben, rechts und unten angegeben wurden.

Die Funktionen liefern jeweils eine Randposition.

9.7.14 Besondere Kommandos senden (SendOK, SendCancel, SendYes, SendNo)

```
SendOK [ ( Fenster ) ]
SendCancel [ ( Fenster ) ]
SendYes [ ( Fenster ) ]
SendNo [ ( Fenster ) ]
```

Mit diesen Kommandos wird ein Druck auf den entsprechenden Button simuliert.

Wird kein Fenstertitel angegeben, wird das gerade aktive Fenster verwendet.

Es ist nicht garantiert, dass diese Befehle bei jedem Programm funktionieren, da die Programmierer nicht die hierfür verwendeten Standard-Signale verwenden müssen.

9.7.15 Fortgeschrittene Kommandos und Mitteilungen senden (SendCommand, SendMessage, PostMessage)

```
SendCommand( [ Fenster, ] Kommando-Id )
PostMessage( [ Fenster, ] Meldungs-Id, wparam, lparam )
SendMessage( [ Fenster, ] Meldungs-Id, wparam, lparam )
Ganz = SendMessage( [Fenster,] Meldungs-Id, wparam, lparam )
```

SendCommand erlaubt es, eine beliebige Kommando-Id zu senden (üblicherweise alle Schaltflächen und Menüeinträge). Dies erfordert aber gute Verbindungen zum Programmierer der betroffenen Anwendung, da diese Ids bei jedem Programm anders sind und sich sogar bei einem Update ändern könnten.

Dasselbe gilt für SendMessage und PostMessage, die eine etwas allgemeingültigere Variante sind. Auch hier sind sehr gute Verbindungen zum Anwendungsprogrammierer nötig. Es kann große Probleme geben, wenn hier unerwartete Daten gesendet werden – üblicherweise Programmabstürze.

Mit SendMessage wird die Meldung vom Ziel-Programm schnellstmöglich bearbeitet und das Script wartet, bis sie bearbeitet wurde. Dadurch ist es auch möglich, einen Rückgabewert vom Programm zu bekommen (Funktions-Variante). Bei PostMessage wird die Meldung an evtl. wartende andere Meldungen angehängt. Das Script läuft sofort weiter, die Meldung wird vom Programm irgendwann später bearbeitet.

MortScript unterstützt nur numerische Parameter.

9.8 Tastendrücke

9.8.1 Zeichenfolgen senden (SendKeys)

SendKeys([*Fenster*,] *Zeichen* **)**

Schickt die angegebene Zeichenfolge an das angegebene bzw. aktuelle (falls kein Fenster angegeben wurde) Fenster.

Beispiele:
```
SendKeys( "Mein Fenster", "Hallo, wie geht's?" )
SendKeys( "Ein Text" )
```

9.8.2 Sonderzeichen (z.B. Richtungstasten) senden (Send...)

SendSpecial(*Tasten-Name* [, *Status*]] **)**

Simuliert den angegebenen Tastendruck.

Derzeit mögliche Tasten sind: „Alt“, „Ctrl“ (Strg), „Shift“ (Großschrit), „CR“ (neue Zeile), „Win“ (Windows-Taste), „Context“ (Kontextmenü), „Tab“, „ESC“, „Space“ (Leerzeichen), „Up“ (hoch), „Down“ (runter), „Left“ (links), „Right“ (rechts), „Home“ (Pos1), „End“ (Ende), „PageUp“ (Seite hoch), „PageDown“ (Seite runter), „Delete“ (Entf), „Backspace“ (←), „Insert“ (Einfg.), „Snapshot“ (Drucken), „F1“ - „F12“, „LeftSoft“ (linke Displaytaste), „RightSoft“ (rechte Displaytaste).

Der Name der Taste muss als Zeichenfolge angegeben werden (also in Anführungszeichen oder als anderes Ergebnis eines Ausdrucks). Groß- und Kleinschreibung werden nicht unterschieden (d.h. “Esc” und “ESC” sind das Gleiche). Nicht alle Tasten funktionieren auf allen System, z.B. gibt es die Displaytasten auf dem PC nicht.

Es ist auch möglich, numerische Tastencodes anzugeben. Diese sind jedoch stark vom Gerät abhängig (v.a. von der Lokalisierung des Systems) und eher für „fortgeschrittene Hacker“ geeignet.

Wenn *Status* nicht angegeben wird, wird die Taste kurz gedrückt und dann wieder losgelassen. Man kann alternativ die Status „down“ (drücken) und „up“ (loslassen) mitgeben, um die Tasten gedrückt zu halten bis sie wieder losgelassen werden sollen (sinnvoll für Alt, Ctrl, Shift und Win). Achte darauf, das „up“ nicht zu vergessen, sonst gibt es hinterher seltsame Effekte...

Send*Sonderzeichen* [**(** *Fenster* [, *Strg?*, *Shift?* [, *Alt?*]] **)**]

Aktiviert das angegebene Fenster und sendet das angegebene Sonderzeichen. Wird kein Fenster angegeben, wird das gerade aktive Fenster verwendet.
Strg?, Shift? und *Alt?* sind optionale Schalter. Wird hier TRUE angegeben, wird die entsprechende Taste mit dem Sonderzeichen gesendet.

Es gibt die folgenden Sonderzeichen:
CR.....ein Zeilenumbruch (Carriage Return)
Tab.....Tabulator
Esc....."Escape" (Abbrechen)
Space.....Leerzeichen
Backspace.....Zeichen vor der Marke löschen („←“)
Delete.....Zeichen nach der Marke löschen („Entf“)
Insert.....„Einfg.“ (schaltet meist zwischen Überschreiben und Einfügen um)
Up/Down/Left/Right.....Steuerkreuz in die entsprechende Richtung
Home.....„Pos1“, an den Anfang der Zeile oder des Dokuments
End.....„Ende“, an das Ende der Zeile oder des Dokuments
PageUp/PageDown.....Seite hoch / runter („Bild ↑“ / „Bild ↓“)
LeftSoft/RightSoft....„Displaytasten“ bei Smartphones und PPCs ab WM5
Win.....„Windows“-Taste bei Smartphones und PPCs ab WM5 (Startmenü)
Context.....„Kontextmenü“-Taste bei PCs und Smartphones/PPCs ab WM5

Beispiele:
```
SendCR( "ERROR" )
SendDown
SendHome( "",0,1 )
```
(bis zum Zeilenanfang markieren)

9.8.3 Bildschirminhalt in die Zwischenablage (Snapshot)

Snapshot [**(** *Fenster* **)**]

Aktiviert das angegebene Fenster (falls angegeben) und kopiert den Bildschirm in die Zwischenablage ("Print screen"-Funktion vom Betriebssystem, funktioniert evtl. nicht bei jedem Programm).

9.8.4 Strg+Taste senden (SendCtrlKey)

SendCtrlKey ([*Fenster*,] *Zeichen* **)**

Sendet Strg + *Zeichen* an das aktuelle oder angegebene Fenster.

`SendCtrlKey( "v" )` sendet z.B. Strg+V (Einfügen) an das aktuelle Fenster.

Beim Zeichen wird nicht zwischen Groß- und Kleinschrift unterschieden, d.h. "v" und "V" machen dasselbe.

Es muss genau ein Zeichen angegeben werden. Dies kann selbstverständlich auch über einen Ausdruck geschehen (also z.B. eine Variable).

9.9 Antippen ("Mausklicks")

9.9.1 Einfaches Antippen/Klicken (MouseClick)

```
MouseClick( [ Fenster, ] x, y )
RightMouseClick( [ Fenster, ] x, y )
MiddleMouseClick( [ Fenster, ] x, y )
```

Simuliert einen Mausklick bzw. Antippen an der entsprechenden Stelle.

Wird ein Fenster angegeben, sind die Koordinaten relativ zu dessen linken oberen Ecke. Bei Fenstern mit Rahmen (z.B. Meldungsboxen und Fragen) zählt dieser mit.

Wird kein Fenster angegeben, ist die linken oberen Ecke des Bildschirms 0,0.

Right... und Middle... sind nur in der PC-Version vorhanden und simulieren einen Mausklick mit der rechten bzw. mittleren Maustaste.

9.9.2 Doppelklick (MouseDblClick)

```
MouseDblClick( [ Fenster, ] x, y )
RightMouseDblClick( [ Fenster, ] x, y )
MiddleMouseDblClick( [ Fenster, ] x, y )
```

Wie MouseClick, simuliert aber einen Doppelklick.

Right... und Middle... sind nur in der PC-Version vorhanden.

9.9.3 Drücken / Loslassen getrennt (MouseDown/MouseUp)

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

Simuliert das Drücken bzw. Loslassen der Maustaste. Die Parameter sind wie bei MouseClick.

Diese beiden Befehle sollten immer zusammen verwendet werden. Mit ihnen lassen sich z.B. "Tippen und Halten" (mit Sleep dazwischen) oder "Ziehen und Loslassen" (MouseUp an der Ziel-Position) simulieren.

Auch hier sind die Right-/Middle-Varianten nur für den PC verfügbar.

9.10 Warten

9.10.1 Feste Pause in Millisekunden (Sleep)

```
Sleep( Millisekunden )
```

Wartet die angegebene Zeit.

9.10.2 Warte-Meldung mit Countdown / Bedingung (SleepMessage)

```
SleepMessage( Sekunden, Meldung [ , Titel [ , OK möglich?
            [ , Bedingung ] ] ] )
```

Wartet die angegebene Zeit und zeigt dabei eine Meldung.

Wenn bei *OK möglich?* 1 angegeben wurde, kann die Meldung weggeklickt werden, ansonsten ist dies nicht möglich.

Wird eine Bedingung angegeben, wird diese jede Sekunde abgeprüft und der Dialog beendet, sobald sie erfüllt ist.

Beispiel:

```
SleepMessage( 10, "Warte auf PocketWord", "Warten...", 0, \
              wndExists( "Word" ) )
```

9.10.3 Warten auf Fenster (WaitFor / WaitForActive)

Siehe 9.7.8 Warten auf Existenz eines Fensters (WaitFor) und 9.7.9 Warten auf Aktivierung eines Fensters (WaitForActive)

9.11 Zeit

9.11.1 Unix-Zeitstempel (TimeStamp, MakeTimeStamp)

```
Ganz = TimeStamp()
Ganz = MakeTimeStamp( Jahr, Monat, Tag
                      [, Stunde [, Minute [, Sekunde ]]] )
```

Gibt die Sekunden seit dem 01.01.1970 zurück.

TimeStamp() liefert hierbei den Wert für die aktuelle Systemzeit, MakeTimeStamp(...) für die angebene Datum/Zeit-Kombination. Werden Zeitangaben weggelassen, wird jeweils „0“ angenommen, d.h., werden alle drei Felder weggelassen, wird Mitternacht verwendet.

9.11.2 Formatierte Ausgabe (FormatTime)

```
Zeichen = FormatTime( Format [, Zeitstempel ] )
```

Gibt die Zeit im Zeitstempel bzw. die aktuelle Zeit, falls keiner angegeben wurde, entsprechend dem angegebenen Format zurück.

Diese Zeichen werden mit dem entsprechenden Wert ersetzt:

H	Stunde (00-23)
h	Stunde (01-12)
a	am/pm
A	AM/PM
i	Minute (00-59)
s	Sekunden (00-59)
d	Tag (01-31)
m	Monat (01-12)
Y	Jahr (4-stellig)
y	Jahr (2-stellig)
w	Wochentag (0=Sonntag bis 6=Samstag)
u	Unix-Zeitstempel
{MM}	Name des Monats (z.B. “Januar”)
{M}	Abgekürzter Name des Monats (z.B. “Jan”)
{WW}	Wochentag ausgeschrieben (z.B. “Montag”)
{W}	Wochentag abgekürzt (z.B. “Mo”)

Alle anderen Zeichen bleiben wie sie waren.

Beachte, dass alle Rückgabewerte Zeichenfolgen sind. Dies ist auch der Fall, um führende Nullen zu erlauben, wie "02" für den Februar, was recht praktisch ist um Datei- oder Verzeichnisnamen zu generieren. Es kann jedoch Probleme verursachen, wenn Arrays verwendet werden sollen. Es müssen dann entweder die Array-Elemente mit entsprechenden alphanumerischen Indizes belegt werden („Monat["01"] = "Erster"”) oder die Zeichenfolge muss in eine Zahl umgewandelt werden, z.B. indem man „FormatTime("m")*1“ verwendet.

Beispiele:

```
x = FormatTime( "H:i:s" )
x = FormatTime( "d.m.Y", TimeStamp() + 86400 )
```

9.11.3 Aktuelle Zeit in mehrere Variablen setzen (GetTime)

```
GetTime( Variable, Variable, Variable )
```

Liest die aktuelle Zeit in drei getrennte Variablen für Stunde, Minute und Sekunden.

```
GetTime( Variable, Variable, Variable,
         Variable, Variable, Variable )
```

Wie oben, aber drei weitere Variablen für Tag, Monat und Jahr (4-stellig).

Beachte, dass alle Rückgabewerte Zeichenfolgen sind. Dies ist der Fall, um führende Nullen zu erlauben, wie "02" für den Februar, was recht praktisch ist um Datei- oder Verzeichnisnamen zu generieren.

9.11.4 Aktuelle Zeit/Datum setzen (SetTime, SetDate)

```
SetTime( Stunde, Minute, Sekunde [, Tag, Monat, Jahr ] )
SetDate( Tag, Monat, Jahr )
```

Setzt die Systemzeit auf die angegebene Zeit bzw. das angegebene Datum. Bei SetDate wird die Uhrzeit nicht modifiziert.

Hinweis: Bitte verwendet diese Funktion nicht, um Testzeiträume auszuhebeln. Seid ehrlich und bezahlt die Programme, die verwendet werden, oder sucht kostenlose Alternativen.

9.12 Dateien kopieren, umbenennen, verschieben und löschen

9.12.1 Einzelne Datei kopieren (Copy)

Copy(*Quelldatei*, *Zieldatei* [, *Überschreiben?*] **)**

Kopiert die angegebene Datei.

Das Ziel muss den Dateinamen enthalten.

Wenn *Überschreiben?* FALSE oder nicht angegeben ist, werden bereits existierende Dateien nicht überschrieben.

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

9.12.2 Mehrere Dateien kopieren (XCopy)

XCopy(*Quell-Dateien*, *Ziel-Verzeichnis* [, *Überschreiben?*
[, *Unterverzeichnisse?*]] **)**

Kopiert die angegebene Dateien in das angegebene Verzeichnis.

Die Quelle kann Wildcards (* und ?) im Dateinamen enthalten (z.B. "\My documents*.psw", nicht aber "\My **.psw").

Das Ziel muss ein existierendes Verzeichnis sein.

Wenn *Überschreiben?* FALSE oder nicht angegeben ist, werden bereits existierende Dateien nicht überschrieben.

Ist *Unterverzeichnisse?* TRUE, werden alle Dateien, die zum Filter passen, auch aus Unterverzeichnissen kopiert. Die Zielverzeichnisse werden wo nötig angelegt.

Beispiel:
```
XCopy( "\My documents\*.txt", "\Storage" )
XCopy( "\My documents\*.txt", "\Storage", TRUE, TRUE )
```
(kopiert z.B. auch „\My documents\texts\x.txt“ nach „\Storage\texts\x.txt“)

9.12.3 Datei umbenennen oder verschieben (Rename)

Rename(*Quelldatei*, *Zieldatei* [, *Überschreiben?*] **)**

Benennt die angegebene Datei um oder verschiebt sie.

Es muss auch beim Ziel der gesamte Pfad angegeben werden!

Wenn *Überschreiben?* FALSE oder nicht angegeben ist, werden bereits existierende Dateien nicht überschrieben.

9.12.4 Mehrere Dateien verschieben (Move)

```
Move( Quell-Dateien, Ziel-Verzeichnis [, Überschreiben?
      [ , Unterverzeichnisse? ] ] )
```

Verschiebt die angegebenen Dateien in das angegebene Verzeichnis.

Die Quelle kann Wildcards (* und ?) im Dateinamen enthalten (z.B. "\My documents*.psw", nicht aber "\My **.psw").

Das Ziel muss ein existierendes Verzeichnis sein.

Wenn *Überschreiben?* FALSE oder nicht angegeben ist, werden bereits existierende Dateien nicht überschrieben.

Ist *Unterverzeichnisse?* TRUE, werden alle Dateien, die zum Filter passen, auch aus Unterverzeichnissen verschoben. Die Zielverzeichnisse werden wo nötig angelegt.

9.12.5 Datei(en) löschen (Delete)

```
Delete( Dateien )
```

Löscht die angegebene Datei(en).

Die Datei-Angabe kann Wildcards (* und ?) im Dateinamen enthalten (z.B. "\My documents*.psw", nicht aber "\My **.psw").

9.12.6 Dateien auch in Unterverzeichnissen löschen (DelTree)

```
DelTree( Dateien )
```

Löscht die angegebene Datei(en), auch in Unterverzeichnissen.

Ist das (Unter-) Verzeichnis danach leer, wird es ebenfalls entfernt.

Die Datei-Angabe kann Wildcards (* und ?) im Dateinamen enthalten (z.B. "\My documents*.psw", nicht aber "\My **.psw"). Diese werden auch für die Unterverzeichnisse verwendet.

Wenn kein Dateifilter angegeben wurde (z.B. „DelTree("\Temp")) und das Verzeichnis existiert, wird „*.*“ als Filter angenommen.

Bitte mit Vorsicht verwenden!

9.12.7 Verknüpfung/Link erstellen (CreateShortcut)

```
CreateShortcut( Linkdatei, Zieldatei [, Überschreiben? ] )
```

Erstellt eine Verknüpfung auf die angegebene Zieldatei. Dies kann z.B. ein Eintrag im Startmenü sein.

Wenn *Überschreiben?* FALSE oder nicht angegeben ist, wird eine bereits existierende Datei nicht überschrieben.

Beispiel:

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

9.13 Lesen und Schreiben von Dateien

9.13.1 Lesen einer Datei (ReadFile, ReadLine)

```
Zeichen = ReadFile( Dateiname [, Länge [, Kodierung ] ] )
Zeichen = ReadLine( Dateiname [, Kodierung ] )
```

ReadFile liest den gesamten Inhalt einer Textdatei. Die eingelesene Größe wird dabei auf *Länge* oder, wenn diese weggelassen wurde oder 0 ist, 1 MB beschränkt.

ReadLine liest eine einzelne Zeile aus der angegebenen Datei. Der nächste Aufruf von ReadLine liefert dann die nächste Zeile. Wenn es keine Zeilen mehr gibt, wird ein leerer Wert zurück gegeben (siehe 9.2.4 Prüfen ob eine Variable belegt ist (IsEmpty)). Im Gegensatz zu ReadFile funktioniert ReadLine nur mit „normalen“ Dateien, nicht mit seriellen Schnittstellen (siehe auch 9.13.5 Zugriff auf serielle Schnittstellen (SetComInfo)) und Dateien im Internet.

Mögliche Werte für die Kodierung sind entweder die Codepage-Nummern, wenn du sie kennst, wie 1252, 437, usw., oder eine der folgenden Zeichenfolgen:

- "latin1" (Westeuropa)
- "jis" (Japanisch)
- "wansung" (Koreanisch)
- "johab" (Koreanisch)
- "chinesesimp" (Chinesisch, vereinfacht)
- "chinesetrad" (Chinesisch, traditionell)
- "hebrew" (Hebräisch)
- "arabic" (Arabisch)
- "greek" (Griechisch)
- "turkish" (Türkisch)
- "baltic" (baltischer Sprachraum)
- "latin2" (größtenteils Osteuropa)
- "cyrillic" (Kyrillisch)
- "thai" (Thailändisch)
- "utf8" (nur Sonderzeichen, z.B. Umlaute, werden verschlüsselt)
- "unicode" (2 Bytes pro Zeichen, genauer gesagt UTF-16 little endian)
- "utf8-prefix" (wie "utf8", aber die Datei beginnt mit dem Hex-Code EF BB BF, was für manche Editoren und Programme als Kennzeichnung dient)
- "unicode-prefix" (ähnlich "utf8-prefix" aber mit der Kennzeichnung FF FE und als Unicode)

Standard ist die Standardkodierung des Systems. Diese hängt normalerweise von der Lokalisierung von Windows ab. Dateien mit UFT8 oder Unicode werden ebenfalls automatisch erkannt, wenn sie mit den entsprechenden Bytefolgen beginnen (siehe „...-prefix“-Codepages oben)

Ein mit ReadFile eingelesener Inhalt kann dann z.B. mit „ForEach line in split (inhalt, "^LF^", TRUE)“ ausgewertet werden.

Beachte auch die ForEach-Möglichkeiten für INI-Dateien und ReadINI/WriteINI!

Beispiel für 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 Schreiben in eine Datei (WriteFile)

WriteFile(*Dateiname*, *Inhalt* [, *Anhängen?* [, *Kodierung*]] **)**

Schreibt den angegebenen Inhalt in die Datei.

Ist „Anhängen?“ FALSE oder wird der Parameter weggelassen, wird eine evtl. schon vorhandene Datei dabei überschrieben, ansonsten wird der angegebene Inhalt an den vorhandenen angehängt.

Zur Kodierung siehe 9.13.1 Lesen einer Datei (ReadFile, ReadLine).

9.13.3 Lesen eines Werts aus einer INI-Datei (IniRead)

Zeichen = **IniRead(** *Dateiname*, *Abschnitt*, *Schlüssel*
[, *Kodierung*] **)**

Liest einen Eintrag aus einer INI-Datei. Der Abschnitt muss ohne die eckigen Klammern angegeben werden.

Zur Kodierung siehe 9.13.1 Lesen einer Datei (ReadFile, ReadLine).

Beispiel:

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

9.13.4 Schreiben eines Werts in eine INI-Datei (IniWrite)

x = **IniWrite(** *Dateiname*, *Abschnitt*, *Schlüssel*, *Wert*
[, *Kodierung*] **)**

Schreibt einen Eintrag in eine INI-Datei. Der Abschnitt muss ohne die eckigen Klammern angegeben werden.

Zur Kodierung siehe 9.13.1 Lesen einer Datei (ReadFile, ReadLine).

Beachte, dass dies dafür sorgt, dass MortScript die gesamte Datei laden, durchsuchen und wieder schreiben muss. Es kann sein, dass du dies besser selbst erledigst (ReadFile, ForEach mit split, WriteFile) wenn viele Werte verändert werden sollen.

Beispiel:

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

9.13.5 Zugriff auf serielle Schnittstellen (SetComInfo)

```
SetComInfo( Port, Timeout [, Baudrate [, Parität [, Bits
            [, Stoppbits [, Kontrolle ]]]]] )
```

Hiermit wird angegeben, wie auf einen COM-Port zugegriffen werden soll.
Die Funktion muss vor ReadFile oder WriteFile aufgerufen werden. Beim Aufruf dieser Funktionen mit „COM1:“ (oder jedem anderen COM-Port) als Dateiname wird der Zugriff mit den angegebenen Werten initialisiert.

Parameter:

Port.....Der Port als DOS-Dateiname, also z.B. "COM1:". Bitte Großschrift und Doppelpunkt beachten!

Timeout.....Zeitspanne in Millisekunden, nach der das System den Zugriff abbrechen soll

Baudrate.....Die Zugriffsgeschwindigkeit. Üblich sind meist 9600, 14400 oder 56700, Standard beim Weglassen des Parameters ist 9600

Parität.....Parität eines Prüfbits. Möglich sind "none" (keins), "even" (gerade), "odd" (ungerade), "mark" und "space". Üblich ist meist "none", das auch der Standard ist, selten noch "even" oder "odd".

Bits.....Anzahl der Bits pro übertragenem Byte. Heutzutage fast immer 8 (Standard), nur selten sind noch 7 in Verwendung, weniger werden so gut wie nie verwendet.

Stoppbits.....Anzahl der Stoppbits (Pause zwischen den Bytes). Mögliche Werte sind 1 (Standard) 1,5 ("1.5" als Zeichenfolge angeben!) oder 2.

Kontrolle.....Art der Flusskontrolle. Möglich sind "None" (keine), "RTS/CTS" (Standard) und "XON/XOFF".

Hinweis: Je nach System, Treibern und Gerät werden nicht alle Parameter immer korrekt verwendet. Insbesondere der Timeout scheint überall anders interpretiert zu werden, teilweise wird er auch komplett ignoriert.

9.14 Dateisystem-Informationen

9.14.1 Prüfen ob eine Datei oder ein Verzeichnis existiert (FileExists/DirExists)

```
Bool = FileExists( Dateiname )
Bool = DirExists( Verzeichnis )
```

Gibt TRUE zurück, wenn die Datei bzw. das Verzeichnis existiert, FALSE wenn nicht.

Es wird auch FALSE zurückgegeben, wenn der Eintrag nicht dem gefragten Typ entspricht, d.h. „FileExists("\Windows")“ ist „falsch“, weil es sich um ein Verzeichnis handelt.

9.14.2 Freien Speicherplatz feststellen (FreeDiskSpace)

```
Ganz = FreeDiskSpace( Verzeichnis [, Einheit ] )
```

Gibt den freien Speicherplatz im angegebenen Verzeichnis zurück. Standardmäßig geschieht dies in Bytes, eine andere Einheit kann als Parameter angegeben werden. Möglich sind BYTES, KB, MB, oder GB (Konstanten, also ohne Anführungszeichen). Der Rückgabewert liegt bei maximal 2147483147, was bei Bytes etwa 2GB entspricht, 2TB für KB, usw.

Bei Windows Mobile-Geräten wird dabei das Verzeichnis (z.B. „\Storage“ für die Speicherkarte), beim PC das Laufwerk („D:\...“) berücksichtigt.

9.14.3 Größe des Speichermediums feststellen (TotalDiskSpace)

```
Ganz = TotalDiskSpace( Verzeichnis [, Einheit ] )
```

Gibt die Größe des Mediums zurück, auf dem das angegebene Verzeichnis liegt. Standardmäßig geschieht dies in Bytes, eine andere Einheit kann als Parameter angegeben werden. Möglich sind BYTES, KB, MB, oder GB (Konstanten, also ohne Anführungszeichen). Der Rückgabewert liegt bei maximal 2147483147, was bei Bytes etwa 2GB entspricht, 2TB für KB, usw.

Bei Windows Mobile-Geräten wird dabei das Verzeichnis (z.B. „\Storage“ für die Speicherkarte), beim PC das Laufwerk („D:\...“) berücksichtigt.

Ich weiß, 2GB sind nicht viel, aber größere Zahlen kann ein 32Bit-System nicht ohne größere Tricksereien handhaben. (Ja, 4GB wären theoretisch auch möglich – aber dann würde MortScript generell keine negativen Zahlen mehr handhaben können).

9.14.4 Dateigröße ermitteln (FileSize)

```
Ganz = FileSize( Dateiname [, Einheit ] )
```

Gibt die Größe der Datei in Bytes zurück. Standardmäßig geschieht dies in Bytes, eine andere Einheit kann als Parameter angegeben werden. Möglich sind BYTES, KB, MB, oder GB (Konstanten, also ohne Anführungszeichen). Der Rückgabewert liegt bei maximal 2147483147, was bei Bytes etwa 2GB entspricht, 2TB für KB, usw.

9.14.5 Zeitpunkt der Dateierstellung feststellen (FileCreateTime)

Ganz = **FileCreateTime(** *Dateiname* **)**

Liefert den Zeitpunkt, zu dem die Datei erstellt wurde, als Unix-Zeitstempel, oder 0 wenn die Datei nicht existiert.

Siehe auch 9.11 Zeit für Informationen, wie man diesen Wert vergleichen und zur Anzeige formatieren kann.

9.14.6 Zeitpunkt der letzten Änderung feststellen (FileModifyTime)

Ganz = **FileModifyTime(** *Dateiname* **)**

Liefert den Zeitpunkt, zu dem die Datei zuletzt geändert wurde, als Unix-Zeitstempel, oder 0 wenn die Datei nicht existiert.

Siehe auch 9.11 Zeit für Informationen, wie man diesen Wert vergleichen und zur Anzeige formatieren kann.

9.14.7 Dateiattribute ermitteln (FileAttribute)

Bool = **FileAttribute(** *Dateiname*, *Attribut* **)**

Ermittelt den aktuellen Wert des angegebenen Dateiattributs. TRUE=gesetzt, FALSE=nicht gesetzt.

Mögliche Werte für „*Attribut*“:

- `directory` (ist das angegebene Ziel ein Verzeichnis?)
- `hidden` (versteckte Datei?)
- `readonly` (Schreibschutz?)
- `system` (Systemdatei?)
- `archive` (noch nicht gesichert?)

Jeweils als Zeichenfolge (also in Anführungzeichen oder als Inhalt einer Variablen).

9.14.8 Dateiattribute setzen (SetFileAttribute, SetFileAttribs)

SetFileAttribute(*Dateiname*, *Attribut*, *Setzen?* **)**

Setzt (*Setzen?*=TRUE) bzw. entfernt (*Setzen?*=FALSE) das angegebenen Dateiattribut. Alle anderen Attribute bleiben unberührt.

Mögliche Werte für „*Attribut*“:
- `hidden` (versteckte Datei?)
- `readonly` (Schreibschutz?)
- `system` (Systemdatei?)
- `archive` (noch nicht gesichert?)

Jeweils als Zeichenfolge (also in Anführungzeichen oder als Inhalt einer Variablen).

Beispiele:
```
SetFileAttribute("\Test.txt", "hidden", TRUE)
```
→ versteckt die Datei

```
SetFileAttribute("\Test.txt", "readonly", FALSE)
```
→ entfernt den Schreibschutz

SetFileAttribs(*Dateiname*, *Schreibschutz?* [, *Versteckt?* [, *Archivieren?*]])

Setzt die angegebenen Dateiattribute. Mit TRUE wird das Attribut gesetzt, mit FALSE wird es entfernt. Eine leere Zeichenfolge ("") lässt das entsprechende Attribut unverändert.

Beispiele:
```
SetFileAttribs("\Test.txt", "", TRUE )
```
→ versteckt die Datei, lässt den Schreibschutz (und andere Attribute) unverändert

```
SetFileAttribs("\Test.txt", FALSE )
```
→ entfernt den Schreibschutz, lässt alle anderen Attribute unverändert

9.14.9 Versionsnummer ermitteln (FileVersion / GetVersion)

```
x = FileVersion( Dateiname )
GetVersion( Dateiname, Variable, Variable, Variable, Variable )
```

Ermittelt die in der Datei abgelegte Versionsnummer.

Diese Versionsnummer muss nicht immer vorhanden oder gut gepflegt sein. Wenn sie enthalten ist, besteht sie aber immer aus vier Teilen.

Üblich ist die Aufteilung in Major (vor dem Punkt), Minor (nach dem Punkt), Patch (oft Buchstaben, weitere Punkte oder Nachkommastellen) und Build (interner Zähler).

Bei der Funktion FileVersion werden die Teile mit Punkt kombiniert zurückgegeben (z.B. "3.1.2.0"), mit dem Kommando GetVersion wird jeder Teil einer eigenen Variablen zugewiesen.

9.14.10 Dateien/Verzeichnisse in einem Verzeichnis (DirContents)

```
Array = DirContents( Dateien, Typ )
```

Liefert einen Array, der die Dateien und/oder Verzeichnisse im angegebenen Verzeichnis zurück liefert. Die Elemente bestehen nur aus dem Datei- bzw. Verzeichnisname ohne das angegebene Verzeichnis (also z.B. „x.txt“, nicht „C:\Verzeichnis\x.txt“).

Dateien ist eine Kombination aus Pfad und einem Dateifilter, z.B. "\Windows*.exe".

Typ kann DC_FILES, DC_DIRS oder DC_ALL sein (Konstanten, also ohne Anführungszeichen!). Mit DC_FILES werden die enthaltenen Dateien zurückgegeben, mit DC_DIRS die Verzeichnisse, und mit DC_ALL beides.

9.15 ZIP-Archive

9.15.1 Wichtige Hinweise

Mit den mir zur Verfügung stehenden Funktionen ist es nicht möglich, Dateien in einem Archiv zu überschreiben. Wird eine bereits vorhandene Datei nochmal zu einem Archiv hinzugefügt, wird sie wirklich nochmal hinzugefügt, d.h., es gibt zwei Einträge für die gleiche Datei. Nicht alle Packer kommen mit so etwas zurecht. Ich empfehle daher, im Zweifelsfall ein neues Archiv anzulegen.

Ein weiteres Problem ist die Codierung der Dateinamen in ZIP-Archiven. Dafür gibt es leider keine Vorgaben und Unicode wird nicht unterstützt.
MortScript verwendet hier wie die meisten Windows/DOS-Programme die DOS-Codepage 437. Dies kann bei Sonderzeichen und Fremdsprachen (z.B. Kyrillisch oder Griechisch) im Dateinamen zu Problemen führen.

9.15.2 Einzelne Datei packen (ZipFile)

ZipFile(*Quelldatei*, *ZIP-Datei*, *Dateiname im Archiv* [, *Rate*] **)**

Fügt die angegebene Datei zum Archiv hinzu. Die „Quelldatei“ (die zu packende Datei) und die „ZIP-Datei“ müssen dabei mit dem kompletten Pfad angegeben werden, der Dateiname im Archiv ist dagegen üblicherweise ohne oder mit einem relativen Pfad.

Die Komprimierungsrate kann zwischen 1=Speichern und 9=beste liegen, der Standard wenn sie weggelassen wird ist 8.

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

9.15.3 Mehrere Dateien packen (ZipFiles)

```
ZipFiles( Quelldateien, ZIP-Datei [, Unterverzeichnisse?
          [ , Pfad im Archiv [, Rate] ] ] )
```

Fügt die angegebenen Dateien zum Archiv hinzu. Die Quelldateien werden wie bei XCopy oder Move mit einem festen Pfad und Platzhaltern im Dateinamen (z.B. "\My documents*.psw") angegeben.
Wenn „Unterverzeichnisse?“ „1“ ist, werden auch die Unterverzeichnisse berücksichtigt, d.h., "\My documents*.psw" würde auch "\My documents\Word\x.psw" finden.
Im Archiv wird der angegebene Pfad weggelassen und – falls angegeben – der „Pfad im Archiv“ davor gesetzt, d.h., ist kein Pfad angegeben, wird "\My documents\Word\x.psw" im Archiv als "Word\x.psw" abgelegt, "\My documents\x.psw" als "x.psw", usw. Wäre dagegen der „Pfad im Archiv“ z.B. "docs", wären die Dateinamen im Archiv "docs\Word\x.psw" bzw. "docs\x.psw".

Beispiele:

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

→ Packt alle *.psw von \Storage\Test und Unterverzeichnissen in das Verzeichnis „test“ im Archiv \Storage\mans.zip

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

→ Packt alle *.jpg von \Storage\Test in das Hauptverzeichnis vom Archiv \Storage\jpgs.zip. Unterverzeichnisse werden nicht durchsucht.

9.15.4 Einzelne Datei entpacken (UnzipFile)

```
UnzipFile( ZIP-Datei, Dateiname im Archiv, Zieldatei )
```

Entpackt die angegebene Datei.
Die Zieldatei muss mit dem kompletten Pfad angegeben werden, der Pfad im Archiv wird dabei nicht berücksichtigt.

Beispiel:

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

→ Entpackt die Datei „test\test.psw“ des Archivs „\Storage\mans.zip“ nach „\Storage\test.psw“

9.15.5 Gesamtes Archiv entpacken (UnzipAll)

```
UnzipAll( ZIP-Datei, Zielverzeichnis )
```

Entpackt alle enthaltenen Dateien des Archivs in das angegebene Verzeichnis. Im Archiv enthaltene Pfade werden dabei verwendet und ggf. angelegt.

9.15.6 Pfad eines Archivs entpacken (UnzipPath)

UnzipPath(*ZIP-Datei, Pfad im Archiv, Zielverzeichnis* **)**

Entpackt alle in einem bestimmten Pfad des Archivs enthaltenen Dateien.
Unterverzeichnisse werden auch entpackt, der angegebene Pfadname wird **nicht** im Zielverzeichnis angelegt. Das Zielverzeichnis muss existieren.

Beispiel:
```
UnzipPath( "\Storage\mans.zip", "test", "\Storage\test-unzip" )
```
→ Entpackt alle Dateien im Verzeichnis „test“ und Unterverzeichnissen davon des Archivs „\Storage\mans.zip“ nach „\Storage\test-unzip“. „test\sub\x.psw“ würde also nach „\Storage\test-unzip\sub\x.psw“ entpackt.

9.16 Verbindungen

9.16.1 Verbindung aufbauen (Connect)

```
Connect
Connect( Verbindungsname )
Connect( Titel, Meldung )
```

Baut eine Verbindung zum Internet auf.

Connect ohne einen Parameter versucht dabei, die Standardverbindung zu verwenden, was aber leider zumindest auf PPCs nicht sehr zuverlässig funktioniert.

Connect mit einem Verbindungsnamen verwendet die angegebene Verbindung. Die Namen können dabei in den Systemeinstellungen frei konfiguriert werden und sind meist vom Netzanbieter vorbelegt. Für die Internetverbindung werden meist „Internet“, „The Internet“ oder ähnliche Namen verwendet.

Werden als Parameter Titel und eine Meldung angegeben, werden alle möglichen Verbindungen in einer Auswahl wie bei Choice aufgelistet und die Ausgewählte verwendet.

Nicht verfügbar auf: PC, PNA

9.16.2 Verbindung beenden (CloseConnection/Disconnect)

```
CloseConnection
Disconnect
```

CloseConnection gibt eine Verbindung frei, die mit Connect aufgebaut wurde. Dies signalisiert dem System nur, dass MortScript nicht mehr damit arbeiten wird. Es ist dem System überlassen, wie es darauf reagiert, d.h., die Verbindung kann – z.B. bei GPRS-Verbindungen – weiterhin bestehen bleiben.

Mit Disconnect werden dagegen alle Verbindungen beendet, auch ActiveSync. Leider funktioniert das seit WM5 AKU3 nicht mehr. Derzeit ist keine Möglichkeit, eine Verbindung von einem Programm aus zu trennen, bekannt.

Nicht verfügbar auf: PC, PNA

9.16.3 Verbindung prüfen (Connected/InternetConnected)

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

Connected prüft ab, ob eine sogenannte RAS-Verbindung („Remote Access“) besteht. Dies ist bei ActiveSync immer, bei Internetverbindungen auf den meisten Geräten der Fall. Liefert TRUE wenn die Verbindung existiert, FALSE wenn nicht.

InternetConnected prüft ab, ob eine Verbindung zum Internet vorhanden ist. Leider liefern hier die meisten Geräte beim reinen Verbindungscheck immer „wahr“ (d.h., die Funktion liefert TRUE) und prüfen die Verbindung erst dann, wenn eine konkrete Adresse angesprochen wird. Deshalb kann optional noch eine URL angegeben werden, die versuchsweise geöffnet wird (z.B. "http://www.google.com").

Nicht verfügbar auf: PC, PNA

9.17 Internet-Zugriff

9.17.1 Proxy vorgeben

```
SetProxy( Proxy )
```

Legt den Proxy für http-Verbindungen fest. Unter Windows Mobile ist es leider nicht so einfach, den Proxy aus den Systemeinstellungen zu verwenden.

Sollte etwas wie "proxy.foo.bar:8080" sein.

Nicht verfügbar auf: PNA

9.17.2 Download (Download)

```
Download( URL, Zieldatei )
```

Lädt die angegebene Datei aus dem Internet herunter.
Die URL muss das Protokoll enthalten ("ftp://..." oder "http://...").
Da dies üblicherweise etwas länger dauert als "Copy" wird hier auch der Fortschritt angezeigt.

Beispiel:

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

Nicht verfügbar auf: Smartphone, PNA

9.17.3 Weitere Möglichkeiten

Alle Dateioperationen, die eine einzelne Datei lesen, funktionieren auch mit einer URL als Quelldatei, also ReadFile, IniRead und die entsprechenden ForEach-Varianten.

9.18 Verzeichnisse

9.18.1 Verzeichnis erstellen (MkDir)

MkDir(*Verzeichnis* **)**

Erstellt das Verzeichnis.

Es ist nicht möglich, mehrere Verzeichnisebenen auf einmal zu erstellen!

D.h., MkDir("\My documents\Some\Path") funktioniert nicht, wenn das Unterverzeichnis "Some" nicht bereits existiert.

9.18.2 Verzeichnis entfernen (RmDir)

RmDir(*Verzeichnis* **)**

Entfernt das Verzeichnis.

Es dürfen keine Dateien oder Unterverzeichnisse im Ordner enthalten sein.

9.18.3 Verzeichnis wechseln (ChDir)

ChDir(*Verzeichnis* **)**

Macht das angegebene Verzeichnis zum aktuellen Verzeichnis.

Nur in der PC-Version enthalten, da Windows Mobile kein „aktuelles Verzeichnis“ kennt.

9.18.4 Systempfade ermitteln (SystemPath)

Zeichen = **SystemPath(** *Typ* **)**

Ermittelt den lokalisierten Verzeichnisnamen für bestimmte Systempfade. Der *Typ* muss als Zeichenfolge angegeben werden, also z.B. "StartMenu".

Mögliche Werte für den „Typ“ sind:
ProgramsMenu.....„Programme“ im Startmenü
StartMenu.....Startmenü, funktioniert nicht bei Smartphones
Startup.....Autostart (Einträge werden nach Softreset gestartet)
Documents....."\My documents" oder Übersetzung, funktioniert nicht mit PPC2002
ProgramFiles....."\Programme" oder Übersetzung, funktioniert nicht mit PPC2002
AppData....."\Application Data" oder Übersetzung
ScriptExe.....Pfad zu MortScript.exe (ohne Dateiname), funktioniert nicht bei Smartphones
ScriptPath.....Pfad zum aktuellen Script (ohne Dateiname)
ScriptName.....Name des aktuellen Scripts (ohne Pfad+Erweiterung)
ScriptExt.....Erweiterung des aktuellen Scripts (".mscr" oder ".mortrun").

D.h., man könnte das aktuelle Script mit folgendem Befehl ausführen:

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

9.19 Registry

9.19.1 Registry-Einträge lesen (RegRead, RegReadExt)

```
Wert = RegRead( Wurzel, Pfad, Eintrag )
Zeichen = RegReadExt( Wurzel, Pfad, Eintrag )
```

Liest den angegebenen Wert aus der Registry.

Für die *Wurzel* gibt es folgende Möglichkeiten:

HKCU.....HKEY_CURRENT_USER
HKLM.....HKEY_LOCAL_MACHINE
HKCR.....HKEY_CLASSES_ROOT
HKUS.....HKEY_USERS
Es sind jeweils nur die vierbuchstabigen Kürzel erlaubt. Seit V4.11 sind auch die Konstanten definiert, so dass keine Anführungszeichen mehr benötigt werden.

Ist der *Eintrag* eine leere Zeichenfolge (""), wird der Standardwert verwendet (in Registry-Editoren üblicherweise mit „<Default>“, „<Standard>“ oder „@“ gekennzeichnet).

Bei RegRead wird der Datentyp wird dabei automatisch berücksichtigt, d.h. DWords werden als Zahl zurückgegeben, Zeichenfolgen als eben diese, binäre Daten als Hexdump in einer Zeichenfolge (z.B. "010ACF") und sogenannte „MultiStrings“ als Array mit Zeichenfolgen-Elementen.

RegReadExt arbeitet ähnlich wie RegRead, gibt den Wert aber so zurück, wie er in Microsofts .reg-Dateien gespeichert würde:

- dword:00000000 für ganze Zahlen (hexadezimal)
- "..." für Zeichenfolgen (mit den Anführungszeichen, enthaltene als \")
- hex:00,00,.... für binäre Daten (Hexdump)
- hex(7):00,00,... für mehrfache Zeichenfolgen („MultiStrings“) (Hexdump)
- hex(2):00,00,... für erweiterbare Zeichenfolgen (Hexdump)

9.19.2 Registry-Einträge schreiben (RegWriteString/-DWord/-Binary/-MultiString, RegWriteExt)

RegWriteString(*Wurzel*, *Pfad*, *Eintrag*, *Wert* **)**
RegWriteDWord(*Wurzel*, *Pfad*, *Eintrag*, *Wert* **)**
RegWriteBinary(*Wurzel*, *Pfad*, *Eintrag*, *Wert* **)**
RegWriteMultiString(*Wurzel*, *Pfad*, *Eintrag*, *Array* **)**
RegWriteExt(*Wurzel*, *Pfad*, *Eintrag*, *Wert* **)**

Schreibt einen Wert in die Registry.

Die gültigen Werte für die *Wurzel* sind bei RegRead aufgelistet.

Ist der *Eintrag* eine leere Zeichenfolge (""), wird der Standardwert verwendet (in Registry-Editoren üblicherweise mit „<Default>“, „<Standard>“ oder „@“ gekennzeichnet).

Bei RegWriteString ist der Wert eine Zeichenfolge.

Bei RegWriteDWord muss der Wert eine Zahl enthalten (bei ungültigen Werten wird eine „0“ geschrieben).

Bei RegWriteBinary muss der Wert eine Zeichenfolge mit dem Hexdump sein (z.B. "010A"), Leerezeichen u.ä. sind darin nicht erlaubt!

Bei RegWriteMultiString muss der Wert ein Array sein. Es werden alle Elemente von 1 bis zur ersten nicht belegten Nummer berücksichtigt, wenn nötig werden sie in Zeichenfolgen umgewandelt.

RegWriteExt ist das Gegenstück zu RegReadExt (siehe oben), es wird ein Wert im .reg-Format benötigt.

Beispiele:

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

(SMS-Benachrichtigung als Default bei PE-Geräten)

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

RegWriteBinary( "HKCU", "Software\Mort\Dummy", "", "C000" )

RegWriteMultiString( "HKCU", "Software\Mort\Dummy", "Tage", \
                     Array( "Mo", "Di", "Mi" ) )

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

9.19.3 Existenz eines Eintrags abfragen (RegValueExists)

`bool = **RegValueExists(** *Wurzel*, *Pfad*, *Eintrag* **)**`

Liefert TRUE zurück, wenn der angegebene Eintrag existiert, FALSE wenn nicht.

Die möglichen Angaben für *Wurzel* und *Eintrag* sind wie in RegRead.

9.19.4 Existenz eines Schlüssels (Pfads) abfragen (RegKeyExists)

`bool = **RegKeyExists(** *Wurzel*, *Pfad* **)**`

Liefert TRUE zurück, wenn der angegebene Schlüssel (ein "Unterverzeichnis" in der Registry) existiert, FALSE wenn nicht.

Die möglichen Angaben für *Wurzel* und *Eintrag* sind wie in RegRead.

9.19.5 Datentyp eines Eintrags abfragen (RegType)

`Zeichen = **RegType(** *Wurzel*, *Pfad*, *Eintrag* **)**`

Liefert den Datentyp des Registry-Eintrags.

Die möglichen Angaben für *Wurzel* und *Eintrag* sind wie in RegRead.

Mögliche Rückgabewerte sind „binary“, „dword“ (ganze Zahl), „link“ (sehr selten verwendet), „multi_sz“ (mehrere Zeichenfolgen), „sz“ (Zeichenfolge), „expand_sz“ (Zeichenfolge mit Variablen), „none“ (kein Wert vorhanden) und „unknown“ (neue/benutzerdefinierte Typen).

9.19.6 Registry-Eintrag entfernen (RegDelete)

RegDelete(*Wurzel*, *Pfad*, *Eintrag* **)**

Entfernt den Registry-Eintrag.

Die möglichen Angaben für *Wurzel* und *Eintrag* sind wie in RegRead.

9.19.7 Registry-Schlüssel (Pfad) entfernen (RegDeleteKey)

RegDeleteKey(*Wurzel*, *Pfad*, *Einträge?*, *Unterschlüssel?* **)**

Entfernt einen ganzen Schlüssel (ein „Unterverzeichnis“ in der Registry).

Ist *Werte?* auf 1 gesetzt, werden auch die enthaltenen Werte gelöscht.

Ist *Unterschlüssel?* 1, werden auch darunter liegende Schlüssel entfernt. Dabei wird auch *Werte?* berücksichtigt, d.h. ist *Werte?* 0, werden auch keine Werte in Unterschlüsseln entfernt (also nur leere Schlüssel gelöscht).

Wenn der Schlüssel nicht gelöscht werden kann, wird eine Meldung angezeigt wenn der ErrorLevel auf "warn" oder niedriger steht.

Der Pfad darf nicht leer sein, um ein versehentliches Zerstören der Registry zu vermeiden. Dennoch sollte dieser Befehl mit großer Vorsicht verwendet werden.

9.20 Dialoge

9.20.1 Freie Text-Eingabe (Input)

```
Zeichen = Input( Meldung [, Titel [, Numerisch? [, Mehrzeilig?
                  [, Vorgabe ]]]] )
```

Öffnet einen einfachen Dialog, der es ermöglicht einen Wert einzugeben, der dann von der Funktion zurückgegeben wird.

Ist *Numerisch?* TRUE, können nur Ziffern eingegeben werden.

Ist *Mehrzeilig?* TRUE, wird eine mehrzeilige Textbox angezeigt. Auf den meisten Systemen wird *Numerisch?* ignoriert, wenn diese Option gesetzt ist.

Wird eine *Vorgabe* angegeben, wird der entsprechende Text als Standard vorgegeben.

Beachte, dass der Rückgabewert auch dann eine Zeichenfolge ist, wenn *Numerisch?* TRUE ist.

Siehe auch 9.20.10 Schriftart für große Meldungen setzen (SetMessageFont)

9.20.2 Meldung (Message)

```
Message( Text [, Titel ] )
```

Zeigt den angegebenen Text in einem Meldungsfenster.

9.20.3 Mehrzeilige Meldung mit Scrollleiste (BigMessage)

```
BigMessage( Text, [ , Titel ] )
```

Im Prinzip wie Message, verwendet aber statt des Systemdialogs, der sich automatisch der Größe des enthaltenen Texts anpasst (außer bei Smartphones) einen eigenen Dialog in fester Größe, der den Text mit einer Scrollbar anzeigt. Dies ist z.B. zur Anzeige von Dateiinhalten sinnvoll.

Siehe auch 9.20.10 Schriftart für große Meldungen setzen (SetMessageFont)

9.20.4 Meldung mit Countdown/Bedingung (SleepMessage)

```
SleepMessage( Sekunden, Meldung [ , Titel [ , OK möglich?
            [ , Bedingung ] ] ] )
```

Siehe 9.10.2 Warte-Meldung mit Countdown / Bedingung (SleepMessage) und 9.20.10 Schriftart für große Meldungen setzen (SetMessageFont)

9.20.5 Einfache Abfrage (Question)

```
Ganz = Question( Frage [, Titel [, Typ ] ] )
```

Zeigt eine einfache Abfrage an. Es wird dazu ein Systemdialog verwendet, die Schaltflächen werden deshalb vom System angezeigt und somit von diesem übersetzt.

Mögliche Typen:
"YesNo".....Zeigt „Ja“ und „Nein“ (Standard)
"YesNoCancel".....Zeigt „Ja“, „Nein“ und „Abbrechen“
"OkCancel".....Zeigt „OK“ und „Abbrechen“
"RetryCancel".....Zeigt „Wiederholen“ und „Abbrechen“

Der Typ muss eine Zeichenfolge sein, also entweder in Anführungszeichen ("YesNo") oder z.B. eine entsprechend belegte Variable.

Rückgabewerte:
„Ja“, „OK“, „Wiederholen“: 1 (vordefinierte Variable „YES“)
„Nein“: 0 (vordefinierte Variable „NO“)
„Abbrechen“: 2 (vordefinierte Variable „CANCEL“)

Beachte dabei, dass „Abbrechen“ in einer einfachen (If/While-) Bedingung wie „Ja“ als „erfüllt“ gilt, du musst es also z.B. mit „If (Question(...., "OkCancel") = 2)“ oder „Switch(Question(....))“ abfragen.

9.20.6 Mehrfachauswahl (Choice)

```
Ganz = Choice( Titel, Hinweis, Standard, Zeit, Wert, Wert
               {, Wert } )
Ganz = Choice( Titel, Hinweis, Standard, Zeit, Array )
```

Funktioniert im Prinzip genauso wie 8.4 ChoiceDefault, liefert den gewählten Index aber als Rückgabewert statt eine Kontrollstruktur zu eröffnen.

Switch(Choice(...)) und ChoiceDefault(...) machen also das Gleiche.

Sinnvoll ist Choice als Funktion dann, wenn der Wert erst später oder an verschiedenen, nicht direkt zusammen hängenden Stellen abgefragt wird.

Siehe auch 9.20.9 Größe und Schriftart für Elemente in Auswahldialogen (SetChoiceEntryFormat)

9.20.7 Verzeichnis auswählen (SelectDirectory)

```
Zeichen = SelectDirectory( Titel, Info [, Vorgabe ] )
```

Zeigt einen Dialog, in dem ein Verzeichnis ausgewählt werden kann. Wenn eine *Vorgabe* angegeben wurde und der angegebene Pfad existiert, wird er vorausgewählt.

9.20.8 Datei auswählen (SelectFile)

```
Zeichen = SelectFile( Titel, speichern?, [Filter [,Info
                      [,Vorgabe]]] )
```

Zeigt einen Dialog, in dem eine Datei ausgewählt werden kann.

Wenn *speichern?* TRUE ist, kann der Benutzer auch einen Dateinamen eingeben, ansonsten kann er nur existierende Dateien auswählen.

Der *Filter* ermöglicht es, nur Dateien anzuzeigen, die zu einer bestimmte Maske passen, wie "*.txt" oder "prefs.*".

Die *Info* funktioniert bei der PC-Version anders als bei den anderen Varianten. Auf dem PC wird der Systemdialog verwendet. Da dort keine eigenen Elemente möglich sind (zumindest nicht so, dass es den Aufwand wert wäre), wird der Text in der Beschreibung des Dateityps angezeigt (wo normalerweise etwas wie „Alle Dateien (*.*)“ steht). Da der Dateiauswahl-Dialog von Windows Mobile ziemlich furchtbar ist, wird dort ein eigener Dialog verwendet, der die Info oben anzeigt. Wenn das Script auf beiden Plattform-Arten laufen soll, sollte hier etwas möglichst neutrales gewählt werden, z.B. „Textdatei wählen“.

Wenn eine *Vorgabe* angegeben wurde, wird sie vorausgewählt (wenn *speichern?* FALSE ist, nur wenn sie auch existiert).

9.20.9 Größe und Schriftart für Elemente in Auswahldialogen (SetChoiceEntryFormat)

```
SetChoiceEntryFormat( Eintragshöhe [, Schriftgröße, Schriftname ])
```

Hiermit wird die Höhe der Einträge sowie, falls angegeben, die Schriftart und -größe, in Choice-Auswahlen festgelegt. Dies betrifft alle Choice-Dialoge, die nach dem Aufruf dieses Kommandos angezeigt werden, also sowohl die Choice- und ChoiceDefault-Ablaufstrukturen als auch die Choice-Funktion.

Die *Eintragsgröße* wird in Pixeln angegeben. Auf mobilen Geräten mit VGA wird die Größe – wie auch die Schriftgröße – automatisch verdoppelt.

Die *Schriftgröße* ist in Punkten, der *Schriftname* z.B. „Courier“, „Tahoma“, usw. Bedenke, dass nicht alle Geräte alle Schriftarten installiert haben, „Tahoma“ (für mobile Geräte) oder „Arial“ (für den PC) sind meist eine gute Wahl.

9.20.10 Schriftart für große Meldungen setzen (SetMessageFont)

```
SetMessageFont( Schriftgröße, Schriftname )
```

Hiermit wird die Schriftart und -größe für Dialoge festgelegt, die von MortScript erzeugt werden, also BigMessage, SleepMessage und Input. „Message“ und „Question“ verwenden einen Systemdialog, der es nicht erlaubt eine andere Schrift zu wählen.

Die Parameter entsprechen inhaltlich den Schriftparametern von SetChoiceEntryFormat oben.

9.21 Statusfenster

9.21.1 Was ist das Statusfenster?

Die Dialoge aus 9.20 Dialoge unterbrechen die Abarbeitung des Scripts, d.h., es geht erst dann weiter, wenn z.B. eine Meldung bestätigt wurde. Um „nebenbei“ anzuzeigen, was das Script gerade macht (damit der Benutzer sieht, wie es voran geht, oder zur Fehlersuche) ist das recht unpraktisch.

Deshalb gibt es das Statusfenster. In diesem können Meldungen angezeigt werden, ohne dass die Ausführung des Scripts unterbrochen wird. Es kann in zwei verschiedenen Stilen angezeigt werden: entweder wird nur die letzte Meldung groß angezeigt (ähnlich BigMessage) oder eine Liste der letzten Meldungen (ähnlich Choice). Meldungen können auch hinzugefügt und gelöscht werden, ohne dass das Fenster sichtbar ist.

Das Statusfenster wird nur in den Vordergrund geholt, wenn es vorher unsichtbar war oder per Kommando aktiviert wird (9.21.8 Statusfenster anzeigen (StatusShow)). Das Script kann damit auch im Hintergrund arbeiten und Statusmeldungen hinzufügen und löschen, ohne dass die Arbeit des Benutzers durch ständiges Auftauchen eines Fensters gestört wird.

Optional kann das Statusfenster auch einen Cancel-Button zum Abbrechen des Scripts haben und nach dem Beenden des Script offen bleiben (muss dann mit OK geschlossen werden).

9.21.2 Anzeige-Typ festlegen (StatusType)

StatusType(*Stil* [, *offen lassen?* [, *Abbrechen-Button?*]] **)**

Der erste Parameter legt fest, wie das Fenster aussehen soll. Möglich sind:

- ST_HIDDEN – das Fenster wird nicht angezeigt
- ST_LIST – Anzeige der Meldungen als Liste
- ST_MESSAGE – Anzeige der letzten Meldung

Die vorhandenen Meldungen bleiben beim Umschalten des Stils erhalten, z.B. beim Umschalten von ST_LIST auf ST_MESSAGE wird die letzte Meldung in der Liste groß angezeigt.

Ist *offen lassen?* TRUE, bleibt das Fenster offen nachdem das Script beendet wurde. Es wird dann ein OK-Button angezeigt, mit dem es geschlossen werden kann.

Ist *Abbrechen-Button?* TRUE, wird während der Ausführung des Scripts ein „Cancel“-Button angezeigt, mit dem das Script abgebrochen werden kann. Dieser funktioniert ähnlich wie 9.22.9 Script beenden (KillScript), Dateioperationen werden jedoch nicht mitten im Vorgang „abgeschossen“.

9.21.3 Titelzeile und Info-Text festlegen (StatusInfo)

StatusInfo(*Titel* [, *Info*] **)**

Hiermit können Fenstertitel und ein kurzer Infotext, der über der Meldung bzw. der Meldungsliste angezeigt wird, festgelegt werden.

9.21.4 Format für Listeneinträge festlegen (StatusListEntryFormat)

```
StatusListEntryFormat( Eintragshöhe
                       [, Schriftgröße, Schriftart ] )
```

Legt den Stil der Meldungen in der Listenanzeige fest. Die Parameter entsprechen 9.20.9 Größe und Schriftart für Elemente in Auswahldialogen (SetChoiceEntryFormat).

9.21.5 Anzahl der Elemente in der Liste (StatusHistorySize)

```
StatusHistorySize( Anzahl )
```

Hiermit wird festgelegt, wie viele Meldungen in der Liste der Listenansicht verfügbar sind.

Die angegebene Anzahl von Einträgen wird auch gemerkt, wenn das Statusfenster unsichtbar oder in der Meldungsansicht ist. Dies ist z.B. praktisch, wenn erst am Ende ein Protokoll ausgegeben werden soll (Umschalten auf Listenansicht) oder die Meldungen für die Fehlersuche in eine Datei geschrieben werden sollen (siehe 9.21.9 Meldungen im Statusfenster wegschreiben (WriteStatusHistory))

9.21.6 Statusmeldungen hinzufügen (StatusMessage, StatusMessageAppend)

```
StatusMessage( Meldung [ Stil [, offen lassen? [, Abbrechen? ]]] )
StatusMessageAppend( Text )
```

Mit StatusMessage wird eine neue Meldung hinzugefügt. In der Listenansicht wird diese am Ende angehängt und ausgewählt (so dass sie auch sichtbar ist), in der Meldungsansicht ist dann nur diese Meldung sichtbar. Die weiteren Parameter entsprechen 9.21.2 Anzeige-Typ festlegen (StatusType), sie erlauben es, den Stil zusammen mit einer neuen Meldung anzupassen.

Bedenke dabei, dass das Fenster nicht notwendigerweise sichtbar ist (siehe auch 9.21.8 Statusfenster anzeigen (StatusShow)). Für Warnungen u.ä. sind 9.20.2 Meldung (Message) oder 9.20.4 Meldung mit Countdown/Bedingung (SleepMessage) besser geeignet.

Mit StatusMessageAppend wird die letzte Meldung um den angegebenen Text ergänzt. Das ist z.B. sinnvoll für „Wartepünktchen“ oder Erfolgsmeldungen.

Beispiel:

```
StatusMessage( "Schritt 1 ", ST_LIST, TRUE )
For i = 1 to 10
  StatusMessageAppend( "." )
Next
StatusMessageAppend( "OK" )
StatusMessage( "Fertig" )
```

9.21.7 Statusmeldungen löschen (StatusRemoveLastMessage, StatusClear)

```
StatusRemoveLastMessage()
StatusClear()
```

StatusRemoveLastMessage entfernt die letzte Meldung wieder. Das ist z.B. sinnvoll, wenn die vorherige Meldung „überschrieben“ werden soll, also z.B. „Kopiere Dateien“ zu „Dateien kopiert“ werden soll. Bis eine neue Meldung hinzugefügt wird, wird die Meldung davor angezeigt bzw. ausgewählt.

StatusClear entfernt alle bisherigen Meldungen.

Bei beiden Befehlen sollte berücksichtigt werden, dass ein leeres Statusfenster (wenn keine Meldungen vorhanden sind) unschön aussieht. Es sollte also danach ein StatusMessage folgen oder das Fenster unsichtbar sein.

9.21.8 Statusfenster anzeigen (StatusShow)

```
StatusShow()
```

Wie schon in 9.21.1 Was ist das Statusfenster? erwähnt, bleibt das Statusfenster normalerweise im Hintergrund wenn der Anwender ein anderes Programm aktiviert hat und neue Meldungen hinzugefügt werden. Mit diesem Befehl wird es in den Vordergrund geholt.

9.21.9 Meldungen im Statusfenster wegschreiben (WriteStatusHistory)

```
WriteStatusHistory( Dateiname [, Anhängen? [, Kodierung ]] )
```

Hiermit werden alle gemerkten Meldungen (also alles, was in der Listenansicht angezeigt wird/würde) in eine Datei geschrieben.

Die Parameter entsprechen 9.13.2 Schreiben in eine Datei (WriteFile), nur dass der Inhalt natürlich vorgegeben ist.

9.22 Prozesse (laufende Anwendungen)

9.22.1 Wird das Prozesshandling unterstützt? (SupportsProcHandling)

```
Bool = SupportsProcHandling()
```

Gibt TRUE zurück, wenn Funktionen wie ProcExists und Kill auf dem Gerät möglich sind.

Die Funktion ist für PNAs gedacht, funktioniert aber auch auf den anderen Plattformen.

Die Funktionen Kill, ProcExists und ProcList benötigen eine System-Bibliothek (toolhelp.dll), die nicht auf allen „abgespeckten“ Windows CE-Geräten vorhanden ist. Der Aufruf dieser Funktionen würde auf diesen Geräten zu Fehlern führen. Mit dieser Funktion ist es möglich, entweder irgendeine Notlösung einzubauen (z.B. über wndExists und Close) oder eine eigene Fehlermeldung anzuzeigen.

9.22.2 Existenz eines Prozesses abfragen (ProcExists)

```
Bool = ProcExists( Prozessname )
```

Gibt TRUE zurück, wenn der angegebene Prozess läuft, FALSE wenn nicht.

Der „Prozessname“ ist der Name der EXE. Es ist meist besser, ihn ohne Pfad anzugeben, z.B. "solitaire.exe", weil dies schneller ist und weniger fehleranfällig ist (falscher Pfad? Tippfehler? Aus einem anderen Pfad gestartet?). Es ist aber auch möglich, einen Pfad mitzugeben, der dann ebenfalls geprüft wird. Auf dem PC funktioniert eine Abfrage mit Pfad nicht bei allen Programmen, insbesondere bei Diensten ist es oft nicht möglich, den Pfad abzufragen.

9.22.3 Existenz eines Script-Prozesses abfragen (ScriptProcExists)

```
Bool = ScriptProcExists( Scriptname )
```

Gibt TRUE zurück, wenn das angegebene MortScript läuft, FALSE wenn nicht.

„ProcExists“ funktioniert für MortScripts nicht sinnvoll, weil der Prozessname für alle MortScripts „MortScript.exe“ ist.

Der Script-Name kann wahlweise ohne Pfad (z.B. „meinscript.mscr“) oder mit vollständigem Pfad (z.B. „\My Documents\meinscript.mscr“) angegeben werden. Bei der PC-Version muss auch das Laufwerk beim vollständigen Pfad enthalten sein.

Siehe auch die Informationen bei 9.22.9 Script beenden (KillScript).

9.22.4 Liste aller laufenden Prozesse (ProcList)

```
Array = ProcList( [ mit Pfad? [, Suchzeichenfolge ] ] )
```

Liefert einen Array mit allen laufenden Prozessen.

Ist *mit Pfad?* TRUE, wird in den Elementen des Arrays der gesamte Pfad zurückgegeben (z.B. „\Windows\sol.exe“), ansonsten nur der Name des Programms ohne Pfad (z.B. „sol.exe“).

Wird eine Suchzeichenfolge angegeben, werden nur die passenden Prozesse zurückgegeben. Der Vergleich berücksichtigt dabei nur den Prozessnamen ohne Pfad und keine Groß-/Kleinschreibung. „test“ würde also „\pfad\zu\einTest.exe“ liefern, nicht aber „\test\pfad\meine.exe“.

9.22.5 Liste aller laufenden Script-Prozesse (ActiveScripts)

```
Array = ActiveScripts( [ mit Pfad? [, Suchzeichenfolge ] ] )
```

Ähnlich ProcList (siehe oben), liefert aber alle laufenden MortScript-Prozesse zurück. Die Elemente enthalten den Namen des Scripts (z.B. „test.mscr“), nicht „MortScript.exe“.

9.22.6 Prozessnamen des aktiven Fensters ermitteln (ActiveProcess)

```
Zeichen = ActiveProcess( [ mit Pfad? ] )
```

Gibt den Programmnamen des gerade aktiven Fensters zurück.

Ist *mit Pfad?* TRUE, wird der gesamte Pfad zurückgegeben (z.B. „\Windows\sol.exe“), ansonsten nur der Name des Programms ohne Pfad (z.B. „sol.exe“).

Auf dem PC ist es nicht immer möglich, den Pfad abzufragen.

9.22.7 Prozessnamen eines angegebenen Fensters ermitteln (WindowProcess)

```
Zeichen = ActiveProcess( Fenstername [, mit Pfad? ] )
```

Gibt den Programmnamen des angegebenen Fensters zurück.

Ist *mit Pfad?* TRUE, wird der gesamte Pfad zurückgegeben (z.B. „\Windows\sol.exe“), ansonsten nur der Name des Programms ohne Pfad (z.B. „sol.exe“).

Auf dem PC ist es nicht immer möglich, den Pfad abzufragen.

9.22.8 Prozess beenden (Kill)

```
Kill( Prozessname )
```

Beendet die angegebene Anwendung. Als Parameter muss der Name der EXE angegeben werden. Dies kann wahlweise mit oder ohne Pfad gesehen. Wie bei ProcExists sollte die Variante ohne Pfad bevorzugt werden, siehe auch 9.22.2 Existenz eines Prozesses abfragen (ProcExists)

ACHTUNG: Dieser Befehl beendet den Prozess ohne Rücksicht auf Verluste!
Es kann zu Datenverlust, Abstürzen oder Fehlermeldungen kommen.
Wo es möglich ist, sollte Close verwendet werden - das gibt dem Programm die Möglichkeit, sich sauber zu beenden (Dateien speichern/schließen usw.).

9.22.9 Script beenden (KillScript)

KillScript(*Scriptname* **)**

Beendet das angegebene Script. KillScript wartet bis zu 3 Sekunden darauf, dass der gerade abgearbeitete Befehl beendet wird, um Probleme durch „brutal abgewürgte“ Befehle zu vermeiden. Klappt das nicht, wird das Script wie bei Kill beendet.

Der Script-Name kann wahlweise ohne Pfad (z.B. „meinscript.mscr“) oder mit vollständigem Pfad (z.B. „\My Documents\meinscript.mscr“) angegeben werden. Bei der PC-Version muss auch das Laufwerk beim vollständigen Pfad enthalten sein.

Wenn der Scriptname ohne Pfad angegeben wird, kann es sein, dass damit ein Script gefunden wird, das zwar den gleichen Namen hat, aber aus einem anderen Pfad stammt als eigentlich gewünscht war. Deshalb sollte, wo es möglich ist, der Pfad mit angegeben werden, z.B. mit Hilfe von 9.18.4 Systempfade ermitteln (SystemPath).

Bedenke, dass ein Script nicht zweimal gestartet werden kann. Wenn ein Hintergrundscript gestartet und beendet werden soll, ist es oft sinnvoll, ein weiteres Script zu schreiben, das das Hintergrundscript startet (Run) wenn es nicht schon läuft (ScriptProcExist) und ansonsten mit KillScript beendet.

Verwende NICHT RunWait oder CallScript zum Starten des Hintergrundscripts, denn sonst wartet das Starter-Script auf das Ende des Hintergrundscripts und kann kein zweites Mal zum Beenden des Scripts aufgerufen werden.

Beispiel:

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

9.23 Signale

9.23.1 Systemlautstärke (SetVolume, Volume)

```
SetVolume( Wert )
int = Volume()
```

Setzt bzw. liest die aktuelle Systemlautstärke. Erlaubt sind Werte zwischen 0 (aus) und 255 (volle Lautstärke).

Manche Geräte, z.B. der Loox720, runden dabei auf bestimmte Zwischenschritte, die meisten erlauben wirklich 256 Lautstärke-Abstufungen.

9.23.2 WAV-Datei abspielen (PlaySound)

```
PlaySound( WAV-Datei )
```

Spielt die angegebene Datei ab. Die Ausführung des Scripts wird solange angehalten.

9.23.3 Vibrieren (Vibrate)

```
Vibrate( Millisekunden )
```

Lässt das Gerät die angegebene Anzahl Millisekunden vibrieren.
Bei der PC-Version wird statt dessen gepiept.
Bei PPCs wird der Vibrator wie eine LED angesprochen, hat aber leider keine einheitliche Nummer. MortScript geht davon aus, dass er die letzte Nummer hat, was bei den meisten Geräten zuzutreffen scheint. Es kann aber (je nach Gerät) auch sein, dass statt dessen eine LED aufleuchtet oder gar nichts passiert.

9.24 Anzeige / Bildschirm

9.24.1 Farbe an Bildschirmposition ermitteln (ColorAt)

```
Ganz = ColorAt( x, y )
```

Ermittelt die Farbe des Punkts an der angegebenen Stelle.

Zumindest auf manchen Geräten scheint dabei aber die Titelzeile ignoriert zu werden, d.h. es wird die Farbe des dahinter liegenden Heute-Hintergrunds zurückgegeben.

9.24.2 Bildschirmausschnitt in Zeichen umwandeln (ScreenToChars)

```
Array = ScreenToChars( x, y, Breite, Höhe, Farbe
                       [, Hintergrundfarbe? [, Zeichen Vordergrund
                       [, Zeichen Hintergrund ] ] ] )
```

Wandelt einen Bildschirmausschnitt anhand von einer Farbe in ein Textmuster um.

So würde z.B. ein schwarzer Kreis auf dem Bildschirm beim Aufruf von ScreenToChars mit 0 als Farbe (=schwarz) einen Array wie diesen zurückgeben:

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

Es handelt sich also im Prinzip „nur“ um einen vereinfachten ColorAt-Vergleich, nicht um eine Texterkennung. Damit kann z.B. zuverlässiger als mit ColorAt abgeprüft werden, ob ein bestimmter Text bzw. eine bestimmte Grafik angezeigt wird. Allerdings ist zu beachten, dass das Ergebnis bei Verwendung von Cleartype („Weichzeichnen“ von Schriftkanten, eine Systemoption von Windows) u.U. unbrauchbar wird.

Wird *Hintergrundfarbe?* auf TRUE gesetzt, wird jede Farbe außer der angegebenen als Vordergrund angenommen. Ansonsten wird nur die angegebene Farbe als Vordergrund verwendet.

Standardmäßig wird „#“ für den Vordergrund und „_“ für den Hintergrund verwendet. Dies kann mit den beiden *Zeichen*-Parametern geändert werden.

9.24.3 Farbe von RGB-Werten erstellen (RGB)

```
Ganz = RGB( Rot, Grün, Blau )
```

Weist der Variablen einen internen Wert für die Farbe zu.

Die Werte für Rot, Grün und Blau dürfen jeweils zwischen 0 und 255 liegen.

Diese Funktion ist v.a. in Kombination mit ColorAt(...) sinnvoll.

9.24.4 Den roten/grünen/blauen Teil einer Farbe ermitteln (Red, Green, Blue)

```
Ganz = Red( Farbe )
Ganz = Green( Farbe )
Ganz = Blue( Farbe )
```

Dies sind die Gegenstücke zu RGB. Damit wird der rote, grüne bzw. blaue Teil einer Farbe (jeweils 0 - 255), die mit ColorAt oder RGB ermittelt wurde, ermittelt.

9.24.5 Bildschirm drehen (Rotate)

```
Rotate( Ausrichtung )
```

Dreht den Bildschirm.
Gültige Werte sind: 0=Standard, 90=rechtshändig, 180=auf dem Kopf, 270=linkshändig

Nicht verfügbar auf: Smartphone, PC, PPC/PNA mit WM2003 oder älter

9.24.6 Hintergrundbeleuchtung ändern (SetBacklight)

```
SetBacklight( Akku, Extern )
```

Setzt die Helligkeit der Hintergrundbeleuchtung auf die angegebenen Werte.
„Akku“ gilt beim Akkubetrieb, „Extern“ bei externer Stromversorgung.
Erlaubt sind Werte zwischen 0 (aus) und 100.
Dieser Befehl funktioniert nicht auf allen Geräten!
Auch der Wert für die höchste Helligkeitsstufe ist vom Gerät abhängig. Mir sind bisher 10, 60 und 100 bekannt, manche arbeiten auch anders herum (z.B. 10=aus, 0=hell) oder haben eine Ausnahme für die stärkste Stufe (z.B. 0=hellste, 1=dunkelste, 10=zweithellste).

Nicht verfügbar auf: PC, PNA, Smartphone

9.24.7 Bildschirm an-/abschalten (ToggleDisplay)

```
ToggleDisplay( Einschalten? )
```

Schaltet den Bildschirm aus (*Einschalten?* = FALSE) oder ein (*Einschalten?* = TRUE).

Nicht verfügbar auf: PC, PNA, Smartphone

9.24.8 Bildschirmgröße abfragen (ScreenWidth, ScreenHeight)

```
Ganz = ScreenWidth()
Ganz = ScreenHeight()
```

Liefert die Breite (ScreenWidth) bzw. Höhe (ScreenHeight) des Bildschirms in Pixeln zurück.

9.24.9 Bildschirmdaten abfragen (Screen)

Bool = **Screen(** *Typ* **)**

Liefert TRUE zurück, wenn die entsprechende Abfrage erfüllt ist, TRUE wenn nicht.
Erlaubte Werte für „Typ“:
"landscape" (Querformat)
"portrait" (Hochkant)
"square" (Quadratischer Bildschirm)
"vga" (VGA-Auflösung – egal ob „Standard“ oder „real/true VGA“)
"qvga" (QVGA-Auflösung)

Nicht verfügbar auf: PC

9.24.10 Heute-Bildschirm aktualisieren (RedrawToday)

RedrawToday

Baut den Heute-Bildschirm neu auf. Sinnvoll, wenn Änderungen in der Registry vorgenommen wurden.

Nicht verfügbar auf: PC

9.24.11 „Sanduhr“ anzeigen/ausblenden (ShowWaitCursor/HideWaitCursor)

ShowWaitCursor
HideWaitCursor

Zeigt die „Sanduhr“ an (ShowWaitCursor) bzw. blendet sie wieder aus (HideWaitCursor).

9.24.12 Aktuellen Mauszeiger herausfinden (CurrentCursor)

Zeichenfolge = **CurrentCursor(** [*Fenster*] **)**

Liefert die Art des aktuellen Mauszeigers für das angegebene Fenster. Wenn kein Fenster angegeben wurde, wird das gerade im Vordergrund befindliche verwendet.

Mögliche Rückgabewerte sind "arrow" (Standard-Pfeil), "wait" (Sanduhr), "cross" (Fadenkreuz), "help" (Fragezeichen), "uparrow" (Pfeil nach oben) und "other" (z.B. von der Anwendung definiert).

Diese Funktion gibt nicht unbedingt den Mauszeiger wieder, den der Anwender sieht. Z.B. zeigt Windows Mobile die Sanduhr (oder besser gesagt die rotierende Scheibe) auch an, wenn Anwendungen gestartet werden oder die Anwendung kein Fenster hat. Das Desktop-Windows dagegen kann z.B. auch den „Anwendungsstart“-Zeiger (Pfeil mit kleiner Sanduhr) oder Pfeile zur Größenänderung von Fenstern anzeigen. Es gibt leider keine zuverlässige Methode, diese Mauszeiger abzufragen. Es ist nur möglich, herauszufinden, welchen Mauszeiger ein Fenster gerne vom System angezeigt hätte.

9.24.13 Show/hide input panel (ShowInput/HideInput)

ShowInput
HideInput

Zeigt bzw. versteckt den Eingabebereich (z.B. Bildschirmtastatur).

Nicht verfügbar auf: PC, Smartphone

9.24.14 Set input panel type (SetInput)

SetInput(*input type* **)**

Legt die Eingabemethode fest, z.B. SetInput("Tastatur") oder SetInput("Transcriber").
Wenn das Script veröffentlicht werden soll, bedenke dass die Namen in anderen Lokalisierungen anders heißen können!

Nicht verfügbar auf: PC, Smartphone

9.25 Zwischenablage

9.25.1 Text in Zwischenablage kopieren (SetClipText)

```
SetClipText( Text )
```

Kopiert den angegebenen Text in die Zwischenablage.

9.25.2 Text aus der Zwischenablage holen (ClipText)

```
Zeichen = ClipText()
```

Gibt den Text aus der Zwischenablage zurück.
Voraussetzung dafür ist, dass eine Text-Variante der Daten in der Zwischenablage zur Verfügung steht. Dies ist von der Anwendung abhängig, in der kopiert/ausgeschnitten wurde.

9.26 Hauptspeicher

9.26.1 Freien Hauptspeicher ermitteln (FreeMemory)

```
Ganz = FreeMemory( [ Einheit ] )
```

Gibt den freien Hauptspeicher zurück. Standardmäßig geschieht dies in Kilobyte, eine andere Einheit kann als Parameter angegeben werden. Möglich sind BYTES, KB, MB, oder GB (Konstanten, also ohne Anführungszeichen). Der Rückgabewert liegt bei maximal 2147483147, was bei Bytes etwa 2GB entspricht, 2TB für KB, usw.

Bei Geräten mit einer Windows Mobile-Version unter 5 wird der Gerätespeicher dynamisch zwischen Hauptspeicher für Programme (FreeMemory()) und „RAM-Disk“ (FreeDiskSpace("\")) aufgeteilt.

9.26.2 Größe des Hauptspeichers ermitteln (TotalMemory)

```
Ganz = TotalMemory( [ Einheit ] )
```

Gibt die Größe des Hauptspeicher zurück. Standardmäßig geschieht dies in Kilobyte, eine andere Einheit kann als Parameter angegeben werden. Möglich sind BYTES, KB, MB, oder GB (Konstanten, also ohne Anführungszeichen). Der Rückgabewert liegt bei maximal 2147483147, was bei Bytes etwa 2GB entspricht, 2TB für KB, usw.

9.27 Energieversorgung

9.27.1 Externe Stromversorgung feststellen (ExternalPowered)

```
Bool = ExternalPowered()
```

Liefert 1 zurück, wenn das Gerät extern mit Strom versorgt wird, 0 wenn nicht.

Bei der PC-Version wird immer 1 zurückgegeben, auch wenn es sich um einen Notebook im Akkubetrieb handelt.

9.27.2 Akkustand (BatteryPercentage)

```
Ganz = BatteryPercentage()
Ganz = BackupBatteryPercentage()
```

Liefert den aktuellen Akkustand in Prozent. Bei externer Stromversorgung kann es – je nach Gerät – sein, dass dieser Wert nicht stimmt.

Ähnliches gilt auch für die Backup-Batterie. Dies wird nicht von jedem Gerät unterstützt. Manche Geräte verwenden Kondensatoren, die nicht abgefragt werden können, anderen fehlt schlicht eine Abfragemöglichkeit in Hard- oder Software oder sie haben gar keine Backup-Batterie weil der Hauptakku fest eingebaut ist. Seit WM5 ist die Backup-Batterie nicht mehr nötig, weil alle Daten auch ohne Strom erhalten bleiben. Es hängt vom Gerät ab, was in diesen Fällen zurück gegeben wird.

Bei der PC-Version wird immer 100 zurückgegeben, auch wenn es sich um einen Notebook im Akkubetrieb handelt.

9.27.3 Gerät ausschalten (PowerOff)

```
PowerOff
```

Schaltet das Gerät aus. Nach dem Einschalten wird das Script fortgesetzt. Beachte dabei, dass Zugriffe auf Speicherkarten direkt nach dem Einschalten meist nicht funktionieren. Ein Sleep und/oder While(not FileExists(...)) danach wäre also ggf. sinnvoll...

Nicht verfügbar auf: PC

9.27.4 Ausschalten verhindern (IdleTimerReset)

```
IdleTimerReset
```

Setzt den Leerlauf-Zähler des Systems zurück. Dadurch lässt sich das automatische Ausschalten verhindern (bei Aufruf in einer Schleife) oder hinauszögern.
Leider verwendet das System für die Abdunklung des Bildschirms einen anderen Zähler, der nicht von außen abgefragt oder beeinflusst werden kann. Dies lässt sich damit also nicht verhindern.

Nicht verfügbar auf: PC

9.28 System

9.28.1 System-Version ermitteln (SystemVersion)

Wert = **SystemVersion(** [*Element*] **)**

Liefert die Version des Betriebssystems zurück. Wenn kein oder ein ungültiger Parameter angegeben wurde, wird die Version im Format Major.Minor.Build zurück gegeben, z.B. 5.1.2600 für XP mit SP2 oder 5.1.195 für WM5.

Mit einem Parameter kann man einzelne Teile zurück bekommen. Mögliche Parameter:
"major".....liefert die Hauptversionsnummer (als ganze Zahl)
"minor".....liefert die Unterversionsnummer (als ganze Zahl)
"build".....liefert den Build-Stand (als ganze Zahl)
"platform".....liefert die Plattform, entweder "Win95" (auch bei 98 und Me), "WinNT" (auch bei XP und Vista) oder "WinCE" (Smartphone / PPC / Windows Mobile).

9.28.2 MortScript-Variante ermitteln (MortScriptType)

Zeichen = **MortScriptType()**

Gibt an, welche MortScript-Version benutzt wird. Derzeit gibt es als mögliche Rückgabewerte:
"PPC".....PocketPC
"PC".....PC (Desktop)
"SP".....Smartphone
"PNA".....PocketNavigation (abgespeckte Windows Mobile-Geräte zur Navigation)

9.28.3 MortScript-Version ermitteln (MortScriptVersion, GetMortScriptVersion)

Zeichen = **MortScriptVersion()**
GetMortScriptVersion(*Variable*, *Variable*, *Variable*, *Variable* **)**

Ermittelt die Versionsnummer von MortScript, entweder als Zeichenfolge („a.b.c.d“) oder in einzelne Variablen (als ganze Zahlen).

Wenn die Funktion MortScriptVersion verwendet wird, werden die Teile der Versionsnummer mit Punkten verbunden, mit dem Kommando GetMortScriptVersion werden sie einzelnen Variablen zugewiesen.

9.28.4 Gerät neu starten (Reset)

Reset

Führt einen Soft-Reset durch. Bitte nur in selbst verwendeten Scripts oder nach einer Vorwarnung/Abfrage durchführen.

Nicht verfügbar auf: PC

10 Alte Syntax und Befehle

Die folgende Syntax, Bedingungen und Befehle sind aus Kompatibilitätsgründen auch noch möglich, **sollten aber nicht mehr verwendet werden.**

Sie sind in dieser Anleitung vor allem aufgelistet, um alte Scripts verstehen zu können.

10.1 Alte Syntax

In älteren Versionen war die Syntax noch anders:

Variablen mussten immer in %...% eingeschlossen werden, außer bei Zuweisungen.

Kommando-Parameter wurden nicht in Klammern angegeben; die Parameter waren nicht generell Ausdrücke, sondern wurden wie folgt unterschieden:

- { ... }: Ein Ausdruck
- %...%: Eine Variable
- "...": Eine Zeichenfolge
- alles andere: Eine Zeichenfolge, die bis zum nächsten Komma geht. Umgebene Leerzeichen/Tabulatoren werden entfernt. Bei Kommandos mit Zuweisungen (z.B. GetTime) auch Variablennamen. %...% sort dafür, dass der Variableninhalt als Variablenname verwendet wird.

Beispiel:

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

statt

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

10.2 Alte Bedingungen

Die alte Bedingungs-Syntax ist alternativ zu den Klammern, also z.B. „If wndExists "Fenster"“. Die Parameter sind hier (außer bei expression / { ... }) **keine Ausdrücke und nicht in Klammern** (siehe auch „Alte Syntax“)

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

Alternative Darstellungen für das jetzt übliche **(** ... **)**
Prüft den angegebenen Ausdruck.

equals *Wert1, Wert2*

Ist erfüllt, wenn die beiden Werte gleich sind. Ist üblicherweise nur sinnvoll, wenn wenigstens ein Wert eine Variable ist (z.B. „If equals %x%, 1“).

fileExists *Datei*

Prüft, ob die angegebene Datei exisitert. Wenn der Parameter als Verzeichnis existiert, ist die Bedingung "falsch"!

dirExists *Verzeichnis*

Prüft, ob das angegebene Verzeichnis exisitert. Wenn der Parameter als Datei existiert, ist die Bedingung "falsch"!

procExists *Anwendung*

Prüft, ob die angegebene Anwendung läuft. Als Parameter muss der Name der EXE ohne Pfad angegeben werden (z.B. solitare.exe).

wndExists *Fenstertitel*

Prüft, ob ein Fenster existiert, das den angegebenen Text im Titel hat (Groß-/Kleinschreibung wird berücksichtigt!). "If wndExists Word" ist z.B. wahr, wenn ein Fenster namens "PocketWord" existiert.

wndActive *Fenstertitel*

Ähnelt "wndExists", ist aber nur wahr, wenn das angegebene Fenster aktiv (im Vordergrund) ist.

question *Text*[, *Titel*]

Zeigt einen einfachen Ja/Nein-Dialog mit dem angegenen Text (Ja/Nein wird von Windows lokalisiert). Die Bedingung ist wahr, wenn „Ja“ gewählt wurde.

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

Prüft, ob der Bildschirm im angegebenen Modus ist.
"screen vga" ist immer wahr, wenn ein VGA-Display verwendet wird, egal ob "doppelte Auflösung" (WM2003 SE-Lösung) oder "real VGA" (SE_VGA, OzVGA, ...) verwendet wird.

regKeyExists *Wurzel*, *Pfad*, *Schlüssel*

Ist wahr, wenn der angegebene Registry-Wert (nicht Schlüssel, trotz des Namens) existiert.
Die Parameter sind wie bei den RegWrite.../RegDelete-Anweisungen.

regKeyEqualsDWord *Wurzel*, *Pfad*, *Schlüssel*, *Wert*
regKeyEqualsString *Wurzel*, *Pfad*, *Schlüssel*, *Wert*

Ist wahr, wenn der Wert in der Registry dem angegebenen Wert entspricht.

Jede Bedinung kann mit dem Präfix "not" negiert werden.
"If not screen landscape" entspricht z.B. "If screen portrait", und "If not fileExists \Windows\some.dll" ist erfüllt, wenn die angegebene Datei nicht existiert.

10.3 Alte Befehle

Input(*Variable*, *Numerisch?*, *Meldung* [, *Titel*] **)**
→ Wie „Input“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

SubStr(*Zeichenfolge*, *Start*, *Länge*, *Variable* **)**
→ Wie „SubStr“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetPart(*Zeichenfolge*, *Trennzeichen*, *Kürzen?*, *Index*, *Variable* **)**
→ Wie „Part“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

Find(*Zeichenfolge*, *zu suchende Zeichenfolge*, *Variable* **)**
→ Wie „Find“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

ReverseFind(*Zeichenfolge*, *Such-Zeichen*, *Variable* **)**
→ Wie „ReverseFind“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetRGB(*Rot*, *Grün*, *Blau*, *Variable* **)**
→ Wie „RGB“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

MakeUpper(*Variable* **)**
MakeLower(*Variable* **)**
→ Ähnlich „ToLower“ bzw „ToUpper“, verändert direkt die angegebene Variable

Eval(*Variable*, *Ausdruck als Zeichenfolge* **)**
→ Wie „Eval“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetColorAt(*x*, *y*, *Variable* **)**
→ Wie „ColorAt“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetWindowText(*x*, *y*, *Variable* **)**
→ Wie „WindowText“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetClipText(*Variable* **)**
→ Wie „ClipText“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetActiveProcess(*Variable* **)**
→ Wie „ActiveProcess“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetTime(*Variable* **)**
→ Wie „TimeStamp“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetTime(*Format*, *Variable* **)**
→ Wie „FormatTime“ ohne Zeitstempel, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetActiveWindow(*Variable* **)**
→ Wie „ActiveWindow“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

IniRead(*Datei*, *Abschnitt*, *Schlüssel*, *Variable* **)**
→ Wie „IniRead“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

ReadFile(*Datei*, *Variable* **)**
→ Wie „ReadFile“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetSystemPath(*Pfad*, *Variable* **)**
→ Wie „SystemPath“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

GetMortScriptType(*variable* **)**
→ Wie „MortScriptType“, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

RegReadString(*Wurzel*, *Pfad*, *Schlüssel*, *Variable* **)**
RegReadDWord(*Wurzel*, *Pfad*, *Schlüssel*, *Variable* **)**
RegReadBinary(*Wurzel*, *Pfad*, *Schlüssel*, *Variable* **)**
→ Ähnlich „RegRead“, Datentyp in der Registry muss angegeben werden, Rückgabe in angegebene Variable statt als Rückgabewert einer Funktion

11 Spenden

Nun, natürlich ist das Programmm Freeware, also MUSST Du nichts bezahlen.
Aber wenn Du meinst "Wau, was für ein geniales Programm, ich möchte ihm so gern ein bisschen von meinem Geld geben, um zu zeigen, wie sehr ich es mag!", dann will ich Dich nicht daran hindern.

Gehe auf www.paypal.de, registriere dich oder melde dich an, und schicke das Geld an mort@sto-helit.de.
Meine Bankverbindung gibt's nur auf Nachfrage.

12 Zu tun (Beta-Kommentare)

foreach in regkeys(...)

SelDirectory mit Verzeichnisanlage

Screenshot-Anzeige mit Koordinaten-Rückgabe

abs(x)