MortScript V4.0

(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)

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

MortScript ist Freeware, d.h., es kann ohne Bezahlung verwendet werden. Änderungen sind jedoch nicht erlaubt.

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“).

Über eine kleine (oder große ;)) Spende als „Dankeschön“ und/oder „Entwicklungs-Anreiz“ würde ich mich freuen. Siehe auch 11 Spenden

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 ForEach-, While- oder Repeat-Schleifen
- Einige Systemfunktionen (z.B. Rotation, Lautstärke, Beleuchtung, Reset)

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 9.25.1 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-4.0-*System*.exe (z.B. MortScript-4.0-PPC.exe) aus dem exe-Verzeichnis des Archivs entpacken, auf dem PC ausführen und den Anweisungen folgen...

3.3 CAB-Datei

Diese Installationsform gibt es nur unter Windows Mobile, also nicht für die PC-Version.
Kopiere die MortScript-4.0-*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 ZIP-Datei

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.

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. Wird also z.B. „test="Dies ist ein Test"“ übergeben, wird mit „Message(test)“ im Script „Dies ist ein Test“ ausgegeben.

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 9.20.5 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 also 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.

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 müssen immer mit absoluter Pfadangabe angegeben werden (z.B. `"\path\to\file.ext"` oder `"\some\directory"`), weil Windows Mobile kein "Aktuelles Verzeichnis" kennt.

6.6 Kommentare

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

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“.

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)`, ...).
MortScript unterstützt keine Gleitkomma-Operationen, deshalb sind Dezimalpunkte oder -kommas nicht möglich!

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 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.

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.

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

7.6 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.

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

liste[1] = "a"
liste[2] = "b"
liste[5] = "f"
liste["a"] = "A"
idx = Choice( "Auswahl", "Wähle etwas", 0, 0, liste )
```
→ Hier werden nur „a“ und „b“ zur Auswahl angezeigt.

7.7 Operatoren

7.7.1 Auflistung der möglichen Operatoren

Alle möglichen Operatoren nach Priorität (höchste zuerst):

`()`	Klammern
`NOT`	Negation
`* / MOD`	Multiplikation, Division, Modulo (Rest von Divisionen)
`+ -`	Addition, Subtraktion
`& \`	Verknüpfung von Zeichenfolgen
`> >= < <= = <>`	Numerische Vergleiche
`gt ge lt le eq ne`	Alphanumerische Vergleiche
`AND &&`	Binäres / logisches Und
`OR ||`	Binäres / logisches Oder

7.7.2 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.

7.7.3 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.

7.7.4 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 }
[ 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.
If, Else 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 } }
EndSwitch
```

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

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

Es sind hier nur numerische Vergleiche möglich.

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 } }
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“). Ansonsten ist die Funktionsweise identisch mit Switch. Theoretisch könnte man also auch `Switch( Choice( ... ) )` (Choice als Funktion, siehe 9.19.5 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.

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:

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 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.

```
ForEach Variable in iniSections ( Dateiname )
```

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

```
ForEach Schüssel, Wert in iniKeys ( Dateiname, Abschnitt )
```

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.

```
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 Unterroutinen (Sub / Call)

```
Sub Unterroutine
  { Anweisungen }
EndSub

Call Unterroutine
```

Mit „Call“ läuft das Script an der Zeile mit der „Sub“-Anweisung mit demselben Parameter weiter. Sobald das Ende der Unterroutine (EndSub) erreicht wurde, wird zur Zeile nach „Call“ zurückgesprungen.

Im Gegensatz zu den meisten anderen Anweisungen muss nach „Sub“ ohne Klammern der Name der Unterroutine stehen, Ausdrücke sind hier nicht erlaubt!

Bei „Call“ ist dagegen theoretisch auch die Variante „Call(*Ausdruck*)“ möglich, wobei der Ausdruck einen gültigen Sub-Namen ergeben muss. Üblicherweise ist „Call Unterroutine“ aber schöner als „Call("Unterroutine")“ oder gar „Call(variableDieUnterroutineEnthaelt)“.

Es gibt keine Parameter oder Rückgabewerte, alle Variablen sind global.

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

8.9 Anderes Script als Unterroutine (CallScript)

```
CallScript( MortScript-Datei )
```

Führt das angegebene Script aus als wäre es eine Unterroutine.
Alle Variablen werden global verwendet, d.h. die im aktuellen Script gesetzten Variablen sind auch im aufgerufenen Script vorhanden und die im aufgerufenen Script gesetzten Variablen sind hinterher auch im aufrufenden Script gesetzt.
Das Script muss mit vollständigem Pfad und Dateinamen angegeben werden.

Beispiel:

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

Für den Aufruf von anderen Programmen oder „selbstständigen“ Scripten gibt es eigene Befehle, siehe 9.5 Ausführen von Anwendungen oder Dokumente öffnen.

8.10 Script vorzeitig beenden (Exit)

Exit

Beendet das Script

9 Kommandos und Funktionen

Funktionen werden durch *x* **= Funktion(...)** dargestellt. Selbstverständlich können sie aber auch in komplexeren Ausdrücken 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 belegen („=“ und Set)

Variable **=** *Ausdruck*

Weist der Variablen das Ergebnis des Ausdrucks zu.

Set(*Variable*, *Ausdruck* **)**

Weist der Variablen das Ergebnis des Ausdrucks zu.
Sollte nicht mehr verwendet werden, sondern besser mit *Variable = Ausdruck* geschehen.
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...)

9.3 Operationen mit Zeichenfolgen

9.3.1 Aufteilen in mehrere Variablen/Arrayelemente (Split)

Split(*Zeichenfolge*, *Trennzeichen*, *Kürzen?*, *Variable*
{, *Variable* } **)**

Teilt die Zeichenfolge am Trennzeichen (darf nur ein Zeichen sein!) und weist die Teile den angegebenen Variablen zu.
Überflüssige Variablen bleiben/werden leer (""), Teile für die keine Variable angegeben wurde, werden ignoriert.
Ist "Kürzen?" ein Wert außer Null, werden die Teile umgebende Leerzeichen, Tabulatoren und Zeilenumbrüche entfernt.

Wenn nur eine Variable angegeben wird, wird diese als Array interpretiert, die einzelnen Teile werden also in Arrayvariable[1] bis Arrayvariable[*n*] geschrieben.

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"

9.3.2 Aufteilen und bestimmten Teil zurückgeben (Part)

```
x = 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)

9.3.3 Länge einer Zeichenfolge (Length)

```
x = Length( Zeichenfolge )
```

Gibt die Anzahl der Zeichen in der Zeichenfolge zurück

Beispiele:

```
x = Length( "Dies ist ein Test" )
```

→ x = 17

9.3.4 Teil einer Zeichenfolge (SubStr)

```
x = 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"

9.3.5 Zeichenfolge in einer Zeichenfolge finden (Find)

```
x = Find( Zu durchsuchende Zeichenfolge, Suchzeichenfolge )
```

Gibt die Position des ersten Zeichens zurück, ab dem die Suchzeichenfolge gefunden wurde. 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( "abcdef", "cd" )
```

→ x = 3

```
x = Find( "abcdef", "CD" )
```

→ x = 0 (Groß-/Kleinschreibung wird berücksichtigt!)

9.3.6 Letztes Vorkommen eines Zeichens finden (ReverseFind)

```
x = 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 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"

9.4 Ausdrücke in Zeichenfolgen (Eval)

x = **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

9.5 Ausführen von Anwendungen oder Dokumente öffnen

9.5.1 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" )
```

9.5.2 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!

9.5.3 Anderes Script als Unterroutine (CallScript)

CallScript(*MortScript-Datei* **)**

Ruft ein anderes Script auf, als wäre es eine Unterroutine.
Siehe 8.9 Anderes Script als Unterroutine (CallScript)

9.5.4 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" )
```

9.5.5 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

9.5.6 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

9.5.7 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

9.6 Fenster

9.6.1 Fenster in den Vordergrund bringen (Show)

```
Show( Fenstertitel )
```

Aktiviert das Fenster mit dem enstprechenden Titel.

9.6.2 Fenster in den Hintergrund verschieben (Minimize)

```
Minimize( Fenstertitel )
```

Versteckt (minimiert) das Fenster mit dem enstprechenden Titel.

9.6.3 Fenster schließen / Anwendung beenden (Close)

```
Close( Fenstertitel )
```

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.

9.6.4 Titel des aktiven Fensters ermitteln (ActiveWindow)

```
x = ActiveWindow()
```

Gibt den Titel des gerade aktiven Programms zurück.

9.6.5 Prüfen, ob ein Fenster aktiv ist (WndActive)

```
x = WndActive( Fenstertitel )
```

Gibt 1 zurück, wenn das angegebene Fenster aktiv ist, 0 wenn nicht.
Dabei kann der angegebene Text an jeder beliebigen Stelle im Fenstertitel auftauchen, Groß- und Kleinschreibung werden berücksichtigt. WndActive("top") liefert also 1, wenn der Heutebildschirm ("Desktop") aktiv ist, nicht aber, wenn "Top-Programm“ aktiv ist.

9.6.6 Prüfen, ob ein Fenster existiert (WndExists)

```
x = WndExists( Fenstertitel )
```

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

9.6.7 Warten auf Existenz eines Fensters (WaitFor)

```
WaitFor( Fenstertitel, Sekunden )
```

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

9.6.8 Warten auf Aktivierung eines Fensters (WaitForActive)

```
WaitForActive( Fenstertitel, Sekunden )
```

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

9.6.9 Fenstertitel / Elementinhalt ermitteln (WindowText)

```
x = 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.

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

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

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 Tastendrücke

9.7.1 Zeichenfolgen senden (SendKeys)

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

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

Send*Sonderzeichen* [**(** *Fenstertitel* [, *Strg?*, *Shift?*] **)**]

Aktiviert das angegebene Fenster und sendet das angegebene Sonderzeichen. Wird kein Fenstertitel angegeben, wird das gerade aktive Fenster verwendet.
Strg? und *Shift?* sind optionale Schalter. Wird hier „1“ 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)

9.7.3 Bildschirminhalt in die Zwischenablage (Snapshot)

Snapshot [**(** *Fenstertitel* **)**]

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

9.7.4 Strg+Taste senden (SendCtrlKey)

SendCtrlKey ([*Fenstertitel*,] *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).

9.8 Antippen ("Mausklicks")

9.8.1 Einfaches Antippen/Klicken (MouseClick)

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

9.8.2 Doppelklick (MouseDblClick)

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

Wie MouseClick, simuliert aber einen Doppelklick.
Right... und Middle... sind nur in der PC-Version vorhanden.

9.8.3 Drücken / Loslassen getrennt (MouseDown/MouseUp)

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

9.9 Warten

9.9.1 Feste Pause in Millisekunden (Sleep)

```
Sleep( Millisekunden )
```

Wartet die angegebene Zeit.

9.9.2 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" ) )
```

9.9.3 Warten auf Fenster (WaitFor / WaitForActive)

Siehe 9.6.7 Warten auf Existenz eines Fensters (WaitFor) und 9.6.8 Warten auf Aktivierung eines Fensters (WaitForActive).

9.10 Zeit

9.10.1 Unix-Zeitstempel (TimeStamp)

```
x = TimeStamp()
```

Gibt die Sekunden seit dem 01.01.1970 zurück.

9.10.2 Formatierte Ausgabe

```
x = 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

Alle anderen Zeichen bleiben wie sie waren.

Beispiele:

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

9.10.3 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).

9.11 Dateien kopieren, umbenennen, verschieben und löschen

9.11.1 Einzelne Datei kopieren (Copy)

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

Kopiert die angegebene Datei.
Das Ziel muss den Dateinamen enthalten.
Wenn "Überschreiben?" 0 oder nicht angegeben ist, werden bereits existierende Dateien nicht überschrieben.

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

9.11.2 Mehrere Dateien kopieren (XCopy)

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

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?" 0 oder nicht angegeben ist, werden bereits existierende Dateien nicht überschrieben.

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

9.11.3 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?" 0 oder nicht angegeben ist, werden bereits existierende Dateien nicht überschrieben.

9.11.4 Mehrere Dateien verschieben (Move)

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

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?" 0 oder nicht angegeben ist, werden bereits existierende Dateien nicht überschrieben.

9.11.5 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").

9.11.6 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.
Bitte mit Vorsicht verwenden!

9.11.7 Verknüpfung/Link erstellen (CreateShortcut)

CreateShortcut(*Linkdatei*, *Zieldatei* **)**

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

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

9.12 Lesen und Schreiben von Dateien

9.12.1 Lesen einer Datei (ReadFile)

x = **ReadFile(** *Dateiname* **)**

Liest den gesamten Inhalt der Datei. Es empfiehlt sich daher, diesen Befehl nur bei kleinen Textdateien zu verwenden. Diese können dann z.B. mit Split (Trennzeichen ^LF^) in einzelne Zeilen aufgeteilt werden.

9.12.2 Schreiben in eine Datei (WriteFile)

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

Schreibt den angegebenen Inhalt in die Datei.
Ist „Anhängen?“ Null (0) oder wird der Parameter weggelassen, wird eine evtl. schon vorhandene Datei dabei überschrieben, ansonsten wird der angegebene Inhalt an den vorhandenen angehängt.

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

```
x = IniRead( Dateiname, Abschnitt, Schlüssel )
```

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

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

9.12.4 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.

9.13 Dateisystem-Informationen

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

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

Gibt „1“ („wahr“) zurück, wenn die Datei bzw. das Verzeichnis existiert, „0“ („falsch“) wenn nicht. Es wird auch „0“ zurückgegeben, wenn der Eintrag nicht dem gefragten Typ entspricht, d.h. „FileExists("\Windows")“ ist „falsch“, weil es sich um ein Verzeichnis handelt.

9.13.2 Freien Speicherplatz feststellen (FreeDiskSpace)

```
x = FreeDiskSpace( Verzeichnis )
```

Gibt den freien Speicherplatz im angegebenen Verzeichnis in Bytes zurück (max. 2147483147 = ~2GB).
Bei Windows Mobile-Geräten wird dabei das Verzeichnis (z.B. „\Storage“ für die Speicherkarte), beim PC das Laufwerk („D:\...“) berücksichtigt.

9.13.3 Dateigröße ermitteln (FileSize)

```
x = FileSize( Dateiname )
```

Gibt die Größe der Datei in Bytes zurück.

9.13.4 Zeitpunkt der Dateierstellung feststellen (FileCreateTime)

```
x = FileCreateTime( Dateiname)
```

Liefert den Zeitpunkt, zu dem die Datei erstellt wurde, als Unix-Zeitstempel, oder 0 wenn die Datei nicht existiert.
Siehe auch 9.10 Zeit für Informationen, wie man diesen Wert vergleichen und zur Anzeige formatieren kann.

9.13.5 Zeitpunkt der letzten Änderung feststellen (FileModifyTime)

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

Liefert den Zeitpunkt, zu dem die Datei zuletzt geändert wurde, als Unix-Zeitstempel, oder 0 wenn die Datei nicht existiert.
Siehe auch 9.10 Zeit für Informationen, wie man diesen Wert vergleichen und zur Anzeige formatieren kann.

9.13.6 Dateiattribute ermitteln (FileAttribute)

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

Ermittelt den aktuellen Wert des angegebenen Dateiattributs. 1=gesetzt, 0=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).

9.13.7 Dateiattribute setzen (SetFileAttribute, SetFileAttribs)

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

Setzt (*Setzen?*=1) bzw. entfernt (*Setzen?*=0) 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:
```
SetFileAttribs("\Test.txt", "hidden", 1)
```
→ versteckt die Datei

```
SetFileAttribs("\Test.txt", "readonly", 0)
```
→ entfernt den Schreibschutz

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

Setzt die angegebenen Dateiattribute. Mit 1 (oder einem anderen Wert außer 0) wird das Attribut gesetzt, mit 0 wird es entfernt. Eine leere Zeichenfolge ("") lässt das entsprechende Attribut unverändert.

Beispiele:
```
SetFileAttribs("\Test.txt", "", 1)
```
→ versteckt die Datei, lässt den Schreibschutz (und andere Attribute) unverändert

```
SetFileAttribs("\Test.txt", 0)
```
→ entfernt den Schreibschutz, lässt alle anderen Attribute unverändert

9.13.8 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 ZIP-Archive

9.14.1 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.

9.14.2 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" )
```

9.14.3 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.

9.14.4 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“

9.14.5 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.

9.14.6 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.

9.15 Verbindungen

9.15.1 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

9.15.2 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

9.15.3 Verbindung prüfen (Connected/InternetConnected)

```
x = Connected()
x = 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 „1“ wenn die Verbindung existiert, „0“ 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 „1“) 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

9.16 Internet-Zugriff

9.16.1 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

9.16.2 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

9.16.3 Weitere Möglichkeiten

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

9.17 Verzeichnisse

9.17.1 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.

9.17.2 Verzeichnis entfernen (RmDir)

RmDir(*Verzeichnis* **)**

Entfernt das Verzeichnis.
Es dürfen keine Dateien oder Unterverzeichnisse im Ordner enthalten sein.

9.17.3 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.

9.17.4 Systempfade ermitteln (SystemPath)

x = **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
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

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

ausführen.

9.18 Registry

9.18.1 Registry-Einträge lesen (RegRead)

```
x = RegRead( 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. (Ggf. Anführungszeichen nicht vergessen, sonst würde der Inhalt z.B. der Variable HKCU verwendet!)

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

Der Datentyp wird dabei automatisch berücksichtigt, d.h. DWords werden als Zahl zurückgegeben, Zeichenfolgen als eben diese und binäre Daten als Hexdump in einer Zeichenfolge (z.B. "010ACF").

9.18.2 Registry-Einträge schreiben (RegWriteString/RegWriteDWord/ RegWriteBinary)

```
RegWriteString( Wurzel, Pfad, Eintrag, Wert )
RegWriteDWord( Wurzel, Pfad, Eintrag, Wert )
RegWriteBinary( 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!

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" )
```

9.18.3 Existenz eines Eintrags abfragen (RegValueExists)

```
x = RegValueExists( Wurzel, Pfad, Eintrag )
```

Liefert 1 zurück, wenn der angegebene Eintrag existiert, 0 wenn nicht.
Die möglichen Angaben für „Wurzel“ und „Eintrag“ sind wie in RegRead.

9.18.4 Existenz eines Schlüssels (Pfads) abfragen (RegKeyExists)

```
x = RegKeyExists( Wurzel, Pfad )
```

Liefert 1 zurück, wenn der angegebene Schlüssel (ein "Unterverzeichnis" in der Registry) existiert, 0 wenn nicht.
Die möglichen Angaben für „Wurzel“ und „Eintrag“ sind wie in RegRead.

9.18.5 Registry-Eintrag entfernen (RegDelete)

```
RegDelete( Wurzel, Pfad, Eintrag )
```

Entfernt den Registry-Eintrag.
Die möglichen Angaben für „Wurzel“ und „Eintrag“ sind wie in RegRead.

9.18.6 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.

9.19 Dialoge

9.19.1 Freie Text-Eingabe (Input)

```
x = 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?“ „1“ (TRUE), können nur Ziffern eingegeben werden.
Ist „Mehrzeilig?“ „1“ (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.

9.19.2 Meldung (Message)

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

Zeigt den angegebenen Text in einem Meldungsfenster.

9.19.3 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.

9.19.4 Meldung mit Countdown/Bedingung (SleepMessage)

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

Siehe 9.9.2 Warte-Meldung mit Countdown / Bedingung (SleepMessage)

9.19.5 Einfache Abfrage (Question)

```
x = 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.

9.19.6 Mehrfachauswahl (Choice)

```
x = Choice( Titel, Hinweis, Standard, Zeit, Wert, Wert
            {, Wert } )
x = 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.

9.20 Prozesse (laufende Anwendungen)

9.20.1 Existenz eines Prozesses abfragen (ProcExists)

```
x = ProcExists( Prozessname )
```

Gibt 1 zurück, wenn der angegebene Prozess läuft, 0 wenn nicht.

Der „Prozessname“ ist der Name der EXE ohne Pfad, z.B. "solitaire.exe".

9.20.2 Existenz eines Script-Prozesses abfragen (ScriptProcExists)

```
x = ScriptProcExists( Scriptname )
```

Gibt 1 zurück, wenn das angegebene MortScript läuft, 0 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 9.20.5 Script beenden (KillScript).

9.20.3 Prozessnamen des aktiven Fensters ermitteln (ActiveProcess)

```
x = ActiveProcess()
```

Gibt den Programmnamen des gerade aktiven Fensters zurück.

9.20.4 Prozess beenden (Kill)

```
Kill( Prozessname )
```

Beendet die angegebene Anwendung. Als Parameter muss der Name der EXE ohne Pfad angegeben werden (z.B. solitaire.exe).

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.).

9.20.5 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 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
```

9.21 Signale

9.21.1 Systemlautstärke ändern (SetVolume)

SetVolume(*Wert* **)**

Setzt die Systemlautstärke auf den angegebenen Wert. 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.

Nicht verfügbar auf: PC

9.21.2 WAV-Datei abspielen (PlaySound)

PlaySound(*WAV-Datei* **)**

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

9.21.3 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.

9.22 Anzeige / Bildschirm

9.22.1 Farbe an Bildschirmposition ermitteln (ColorAt)

x = **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.

9.22.2 Farbe von RGB-Werten erstellen (RGB)

x = **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.

9.22.3 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

9.22.4 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

9.22.5 Bildschirm an-/abschalten (ToggleDisplay)

ToggleDisplay(*Einschalten?* **)**

Schaltet den Bildschirm aus (*Einschalten?* = 0) oder ein.

Nicht verfügbar auf: PC, PNA, Smartphone

9.22.6 Bildschirmdaten abfragen (Screen)

x = **Screen(** *Typ* **)**

Liefert 1 zurück, wenn die entsprechende Abfrage erfüllt ist, 0 wenn nicht.
Erlaubte Werte für „Typ“:
"landscape" (Querformat)
"portrait" (Hochkant)
"vga" (VGA-Auflösung – egal ob „Standard“ oder „real/true VGA“)
"qvga" (QVGA-Auflösung)

Nicht verfügbar auf: PC

9.22.7 Heute-Bildschirm aktualisieren (RedrawToday)

RedrawToday

Baut den Heute-Bildschirm neu auf.

Nicht verfügbar auf: PC

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

```
ShowWaitCursor
HideWaitCursor
```

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

9.23 Zwischenablage

9.23.1 Text in Zwischenablage kopieren (SetClipText)

```
SetClipText( Text )
```

Kopiert den angegebenen Text in die Zwischenablage.

9.23.2 Text aus der Zwischenablage holen (ClipText)

```
x = 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.

9.24 Hauptspeicher

9.24.1 Freien Hauptspeicher ermitteln (FreeMemory)

```
x = FreeMemory()
```

Gibt den freien Hauptspeicher in Kilobyte zurück.
Bei Geräten mit einer Windows Mobile-Version unter 5 wird der Gerätespeicher dynamisch zwischen Hauptspeicher für Programme (FreeMemory()) und „RAM-Disk“ (FreeDiskSpace("\")) aufgeteilt.

9.24.2 Größe des Hauptspeichers ermitteln (TotalMemory)

```
x = TotalMemory()
```

Gibt die Größe des Hauptspeicher in Kilobytes zurück.

9.25 Energieversorgung

9.25.1 Externe Stromversorgung feststellen (ExternalPowered)

```
x = 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.

9.25.2 Akkustand (BatteryPercentage)

```
x = BatteryPercentage()
```

Liefert den aktuellen Akkustand in Prozent. Bei externer Stromversorgung kann es – je nach Gerät – sein, dass dieser Wert nicht stimmt.

Bei der PC-Version wird immer 100 zurückgegeben, auch wenn es sich um einen Notebook im Akkubetrieb handelt.

9.25.3 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

9.25.4 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

9.26 System

9.26.1 System-Version ermitteln (SystemVersion)

```
x = 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
"minor"..... liefert die Unterversionsnummer
"build".....liefert den Build-Stand
"platform"..... liefert die Plattform, entweder "Win95" (auch bei 98 und Me), "WinNT" (auch bei XP und Vista) oder "WinCE" (Smartphone / PPC / Windows Mobile).

9.26.2 MortScript-Variante ermitteln (MortScriptType)

```
x = 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)

9.26.3 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.