AutoIt leicht gemacht

Johannes Mitlmeier

3. August 2010

Gewidmet denen, die für Freiheit eintreten.

Copyright

Inhaltsverzeichnis

1 Einführung

1.1 Über dieses Buch

Dieses Buch hat zum Ziel, Ihnen die Skriptsprache AutoIt näher zu bringen. Es soll die zahlreichen Möglichkeiten und Anwendungsbereiche dieser vielseitigen und vergleichsweise leicht erlernbaren Sprache darstellen.

Beispielquelltext wird an geeigneten Stellen abgedruckt und erklärt. So können Sie die Ausführungen direkt am eigenen Rechner nachvollziehen und hoffentlich leichter verstehen.

Die Beispiele und Ausführungen beziehen sich auf die stabile AutoIt-Version 3.3.0.0. Es ist aus derzeitiger Perspektive nicht anzunehmen, dass sich die vorhandenen Funktionen grundlegend ändern. Eher werden neue Funktionen hinzukommen oder bestehende intern verbessert bzw. beschleunigt werden. Deshalb sollten die Informationen nicht allzu schnell veralten.

1.2 Wozu AutoIt?

Es gibt zahlreiche Gründe, die Menschen dazu bewegt haben, sich mit AutoIt zu beschäftigen. Alle diese Gründe haben ihre Berechtigung und geben damit schon einen Hinweis auf die Vielseitigkeit der Sprache.

Der klassische Anwender automatisiert Installationen [ref] und andere administrative Aufgaben mit AutoIt. Genauso gibt es aber auch diejenigen, die sich kleine Alltagstools schreiben, die lästige Klicks ersparen oder beispielsweise Informationen aus dem Internet darstellen [ref ie.au3/ff.au3].

Ein weiterer Einsatzbereich ist die Automatisierung von (Online-)Spielen, bei denen man sich ebenfalls eine Menge Klickarbeit ersparen kann, auch wenn der moralische Aspekt zweifelhaft bleibt [ref und Text].

AutoIt ist aber keinesfalls auf diese beispielhaften Anwendungsmöglichkeiten beschränkt, sondern kann für so viele Ideen genutzt werden, dass es inzwischen schon eine Art *General Purpose Language* darstellt.

1.3 Warum AutoIt?

Gegenüber anderen Lösungen zur Automatisierung zeichnen AutoIt einige Aspekte aus:

	AutoIt	**Andere Tools**
PREIS	**kostenlos**	oftmals hunderte Euro
FLEXIBILITÄT	**sehr hoch**	unterschiedlich
COMMUNITY	**groß und aktiv**	meist weniger beteiligt
PROGRAMMERSTELLUNG	programmierlastig	**oft gutes IDE-Konzept**
ENDERGEBNIS	**echte Win32-Exe-Datei**	meist proprietäre Formate

Tabelle 1.2: AutoIt gegenüber anderen Automatisierungstools

Auch gegenüber anderen Skriptsprachen hat AutoIt einige Vorteile. Dazu gehört sicherlich der klare Ansatz, sich voll auf Windows zu konzentrieren und Altlasten über Bord zu werfen. Damit ist die Sprache konsistent und moderner ausgerichtet als beispielsweise das antiquierte BASIC und viele seiner Nachfolger.

1.4 Alternativen

Als Alternative zu AutoIt ist vor allem AutoHotkey ([link]) zu sehen, das sich zu einer Zeit von AutoIt abgespalten hat, als dieses noch ein OpenSource-Projekt war.

Es hat sich seither in eine etwas andere Richtung entwickelt. Bei AutoHotkey stehen die Funktionen für Maus- und Tastatursteuerung bis heute sehr stark im Vordergrund und sind denen von AutoIt überlegen.

Dagegen ist die Syntax auf dem Stand von AutoIt Version 2 stehen geblieben und deshalb deutlich inkonsistenter und weniger einfach zu lesen. Auch in den generellen Funktionen abseits der Automatisierung kann AutoHotkey seinem Vater nicht ganz das Wasser reichen.

Es ist daher für verspielte Lösungen, die Maus und Tastatur vollständig kontrollieren müssen, eine gute Alternative, andernfalls aber eher nicht.

2 Über AutoIt

2.1 Geschichte

Die Geschiche von AutoIt beginnt 1999. Damals begann der Engländer Jonathan Bennett für seine Arbeitstelle eine Skriptsprache zu entwickeln, die dafür gedacht war, hunderte Computer mit der gleichen Software und Konfiguration auszustatten. Daraus erklärt sich auch der Name „AutoIt“, eine verkürzte Form von „Automate it“ (engl. für „autmatisiere es“). Die oft kolpotierte Schreibweise „AutoIT“ verändert den Sinn des Namens und ist daher *nicht* äquivalent.

AutoIt wurde im Lauft der Zeit immer besser und erreichte schon in Version 2 weite Verbreitung. Nach und nach fanden sich immer mehr fähige C++-Programmierer, die Jonathan Bennett bei der Entwicklung unterstützten. Anfangs war das Programm OpenSource-Software. Doch nachdem sich andere (kostenpflichtige) Automationstools offenbar daran bedient hatten, wurden die besonders kopierwürdigen Teile des Quelltextes wieder zurückgezogen. Deshalb ist das Programm heute nur noch Freeware. Dennoch gibt es einen Teil des Quelltextes auf der Homepage von AutoIt zum Download.

Auf der offiziellen Seite http://www.autoitscript.com gibt es alle Downloads und ein Forum mit Hilfestellungen und tausenden AutoIt-Skripten zum freien Download.

2.2 Features

- Windows- und DOS-Dateien ausführen [ref]
- Tastendrücke simulieren [ref]
- Mausbewegungen und -klicks simulieren [ref]
- Fenster manipulieren [ref]
- Direkt mit Controls in Fenstern interagieren [ref]
- Mit der Zwischenablage arbeiten [ref]
- Mit der Registry arbeiten [ref]

- Einfache Erstellung auch komplexer Benutzeroberflächen (GUIs) [ref]
- Kompilierte Exe-Dateien erstellen, die vollkommen selbständig lauffähig sind [ref]
- Unterstützung aller Kernfunktionen in Windows 95 bis Windows Vista gleichermaßen

Viele weitere Funktionen lassen sich mit UDFs [ref] erreichen. Damit wird der Funktionsumfang nochmals erheblich erweitert. Beispielsweise lassen sich so ZIP-Funktionen, die Fernsteuerung des Internet Explorers, Zugriff auf Hauptspeicheradressen und vieles mehr nachrüsten.

3 Vorbereitungen

3.1 Installation

3.1.1 Versionen

Es gibt von AutoIt immer zwei Entwicklungsstränge. Eine *stabile Version* erscheint in unregelmäßigen, größeren Abständen. Diese Version ist im Wesentlichen als bug-frei anzusehen, auch wenn das natürlich nie zu 100% zu garantieren ist. Sie ist damit für den Produktiveinsatz geeignet.

Die Grundfunktionen (zu erkennen daran, das sie nicht mit einem Unterstrich beginnen) sind schon so lange im Einsatz, dass man sich auf sie verlassen kann. Bei UDFs (mit Unterstrich am Anfang des Namens) ist das selbst in der stabilen Version nicht immer so, aber in den allermeisten Fällen sind diese auch ausreichend getestet.

Daneben gibt es in kürzerem Abstand *Beta-Versionen*, die neue Funktionen enthalten, die aber noch nicht komplett ausgetestet sind, und gelegentlich Bugs enthalten. Meldungen über bereits gefundene Bugs gibt es im Bug-Report-Forum ([link]). Dort müssten Sie sich auch melden, wenn Sie einen Bug gefunden zu haben glauben.

Es wird empfohlen, zum Programmieren die aktuell stabile sowie eine möglichst aktuelle Beta installiert zu haben. So kann man bei Fehlern in der Beta noch mit der stabilen Version kompilieren, aber auch alle Beta-Funktionen nutzen.

3.1.2 Download

Die aktuell stabile Version von AutoIt gibt es unter http://www.autoitscript.com/autoit3/downloads.php – dort laden Sie sich aus der obersten Zeile die Datei hinter „Download AutoIt v3“ herunter.

Die Beta-Version liegt hier unter http://www.autoitscript.com/autoit3/files/beta/autoit/ zum Download bereit. Achten Sie darauf, dass Sie die Datei mit der höheren Versionsnummer herunterladen, da die letzten beiden Betas dort zur Verfügung gestellt werden.

Bitte beachten Sie die Hinweise zur Installationsreihenfolge! [ref]

3.2 Editor

AutoIt-Skripte sind zunächst einmal ganz normale Textdateien. Das bedeutet, dass es ohne Schwierigkeiten möglich ist, seine Quelltexte mit einem einfachen Texteditor wie beispielsweise dem direkt von Windows bereitgestellten „Notepad“ zu verfassen.

Allerdings haben die AutoIt-Entwickler den freien Editor SciTE speziell an AutoIt angepasst („SciTE4AutoIt“), sodass er mit Syntaxhighlighting, Funktions- und Variablenvervollständigung, Kompilieren (F7 für die stabile bzw. Alt+F7 für die Beta-Version) und Ausführen (F5 bzw. Alt+F5) direkt aus dem Editor heraus das Entwickeln gut unterstützt.

Zudem liefert SciTE4AutoIt zahlreiche Tools wie Tidy (Codeeinrückung) [ref], Koda (GUI-Aufbau) [ref] und Makrorekorder mit. Daher sei die Installation dieses Editors unbedingt angeraten.

Herunterladen kann man ihn auf der für SciTE4AutoIt eingerichteten Webseite unter http://www.autoitscript.com/autoit3/scite/.

Bitte beachten Sie die Hinweise zur Installationsreihenfolge! [ref]

Es gibt auch die Möglichkeit AutoIt mit gewissem Komfort mit anderen Editoren zu nutzen. Dazu stehen auf der Entwicklerseite einige Syntaxdefinitionsdateien bereit, die sich einbinden lassen.

Für jEdit pflegt Thorsten Willert unter http://jedit4autoit.thorsten-willert.de/ eine sehr gut an AutoIt angepasste Version.

3.3 Installationsreihenfolge

Die besten Erfahrungen haben wir mit folgender Installationsreihenfolge gemacht:

1. Installation der stabilen AutoIt-Version *(wenn möglich im vorgeschlagenen Pfad)*

2. Installation des SciTE4AutoIt-Paketes für die stabile Version
3. Installation der AutoIt-Beta-Version
4. Installation des SciTE-Updates für die AutoIt-Beta-Version

Bei den SciTE4AutoIt-Installationen sollte die Standardaktion für einen Doppelklick auf die Datei auf „Edit Script“ eingestellt werden, damit man die Skripte bequem zum Bearbeiten aus dem Explorer heraus öffnen kann.

Damit sollte AutoIt fertig eingerichtet und funktionstüchtig sein.

3.4 Hilfe

3.4.1 Mitgelieferte Hilfe

Bei jeder AutoIt-Installation wird im Installationsverzeichnis (wenn Sie diesen Pfad nicht geändert haben, lautet er auf einem deutschen Windows C:\Programme\AutoIt v3) die Hilfe zur jeweiligen Version abgelegt. Auch im Startmenü-Ordner „AutoIt v3“ gibt es einen Link auf die Hilfe.

Die Inhalte dieser Hilfe-Datei sind von hoher Qualität: Zu jedem Befehl gibt es eine exakte Beschreibung der Syntax (gelb unterlegt) [ref auf „Funktionen aufrufen“] – ergänzt durch eine Auflistung und Erklärung aller möglichen Parameter –, allgemeine Hinweise über die Funktion sowie mindestens ein Beispiel. Besonders letzteres verdient Beachtung, da sich daraus die konkrete Verwendung der Funktion ableiten lässt, die gelegentlich aus den theoretischen Angaben nicht deutlich genug wird.

Zusätzlich zu diesen Funktionsbeschreibungen gibt es einige Seiten, die Hintergründe und Grundkonzepte der Programmierung erläutern. Auch deren Lektüre ist eine große Hilfe beim Einstieg in die neue Sprache.

Das chm-Format der Hilfe ermöglicht auch eine Volltextsuche in allen Beschreibungen. Dazu wählen Sie im linken Teil der Ansicht den Reiter „Suche“.

3.4.2 Deutsche Hilfe

Herunterladen kann man die möglichst aktuelle deutsche Hilfe unter http://www.autoit.de/hilfe.

3.4.3 Deutsche Foren

Aufgrund der zunehmenden Resonanz von AutoIt auch unter deutschen Internetnutzern gibt es inzwischen auch einige deutschsprachige Webseiten und Foren, die sich auch oder ausschließlich mit AutoIt beschäftigen. Das wird vor allem dadurch gefördert, dass die Seite http://www.autoitscript.com weitgehend auf Englisch gehalten ist.

Hier eine unvollständige Liste von deutschen Anlaufstellen zu AutoIt:

http://www.autoit.de versammelt die derzeit größte deutsche Community zu AutoIt um sich. Es gibt ein gut besuchtes Forum, zahlreiche Skripte und Projekte (beispielsweise Galenda [ref]) sowie das Tutorial, auf dessen Grundlage dieses Buch entstanden ist. Dazu kommt Syntaxhighlighting für Forumsposts und eine Online-Hilfe zum direkten Nachschlagen der Funktionen aus dem Quelltext heraus. Ein Vorteil der Seite ist, dass sie sich ausschließlich mit AutoIt beschäftigt und das Forum deshalb auf diese besonderen Bedürfnisse zugeschnitten werden kann.

http://www.german-nlite.de/forum.html ist die deutsche Seite zu „nLite“, einem Programm, um sich angepasste Windows-Installationsmedien zu erzeugen. Auch in diesem Zusammenhang entfaltet AutoIt seine Fähigkeiten. Wieder ist es die automatische Installation [ref], die Hauptthema des zugehörigen Unterforums ist. Aufgeteilt in Support- und Skriptforum bietet es die nötigen Möglichkeiten, seine Probleme oder Programme zu posten. Darüber hinaus gibt es wenige Besonderheiten, die aber die freundliche Community auszugleichen vermag.

http://www.windows-unattended.de/forum/ zu guter Letzt beschäftigt sich mit einem ähnlichen Thema wie http://www.german-nlite.de, aber in einer etwas breiteren Fächerung. Der AutoIt-Teil des Forums, der eine lange Tradition hat, soll jedoch zu Gunsten von http://www.german-autoit.de langsam auslaufen.

3.5 Vom Quelltext zur Exe-Datei

Bei AutoIt ist es außerordentlich einfach, aus seinen Quelltexten ausführbare Dateien zu machen. Während der Installation wurden bereits automatisch alle notwendigen Vorkehrungen getroffen. Es wurde eine so genannte „Shell-Extension“ im Explorer registriert, was bedeutet, dass das Kompilieren über das Kontextmenü erfolgen kann.

Die Quelltextdatei können Sie auch über das Kontextmenü erstellen: Wählen Sie den Eintrag „Neu ▷ AutoIt v3 Script“ und vergeben Sie einen aussagekräftigen Namen. Alternativ können Sie auch eine Textdatei erstellen und sie danach so umbenennen, dass sie die Dateiendung „au3“ erhält.

Um diese Datei mit dem Quelltext nun zu kompilieren, klicken Sie diese Datei wie erwähnt mit der rechten Maustaste an und wählen dann den Eintrag „Compile“. Nach

einigen Sekunden ist im gleichen Ordner eine Datei abgelegt, die wie das Skript heißt, aber jetzt die Endung „exe“ trägt. Zu erkennen ist sie leicht am AutoIt-Logo – dem stilisierten Buchstaben A –, das sie ziert.

Für den Fall, dass Sie beispielsweise dieses Logo ändern oder erreichen möchten, dass die Ausgabedatei anders heißt, wählen Sie im Kontextmenü statt „Compile“ den Eintrag „Compile with options“. Daraufhin erscheint ein Fenster, indem Sie die gewünschten Angaben machen können.

Compiler

Ein Compiler verwandelt den Quelltext, den Sie verfasst haben in Anweisungen für einen Prozessor. Daher sind echt kompilierte Programme schnell, denn es gibt keine Zwischenstufe hin zu Prozessorbefehlen. Allerdings muss für unterschiedliche Rechnerarchitekturen jeweils passender Maschinencode erzeugt werden und der Compilerbau ist eine komplexe Wissenschaft.

Interpreter

Ein Interpreter dagegen liest Ihren Quelltext bei jedem Programmstart neu ein und verarbeitet diesen dann zur Laufzeit. Daher ist die Geschwindigkeit, die dabei erzielt werden kann, im Vergleich zu kompiliertem Code geringer. Allerdings ist ein Interpreter leichter zu schreiben und zu erweitern.

AutoIt

AutoIt verbindet die beiden Konzepte gewissermaßen: Es entsteht am Ende eine selbstständig laufende Exe-Datei, die eher kompiliert aussieht. Beim Starten wird aber der AutoIt-Interpreter und das Skript gewissermaßen entpackt und dann der Interpreter auf das Skript angesetzt.

4 Grundlagen

4.1 Skriptausführung

Ein Skript, dass man über das Kontextmenü per „Run“ oder aus SciTE heraus startet, wird an die Datei „AutoIt.exe“ übergeben. Diese Datei ist der AutoIt-Interpreter. Das bedeutet, dass diese Datei den Quelltext von oben nach unten durchgeht und die jeweiligen Befehle ausführt.

Details zu diesem Interpreter-Konzept finden Sie [ref].

4.2 Kommentare

In AutoIt gibt es zwei Typen von Kommentaren. Zum einen existieren einzeilige Kommentare, die mit einem Semikolon eingeleitet werden. Das bedeutet, dass alles, was hinter diesem Zeichen in einer Quelltextzeile steht, nicht mehr ausgeführt wird.

Listing 4.1: Einzeilige Kommentare

```
1 ; Das sind einige Beispielkommentare
2 Sleep(2000) ; zwei Sekunden Pause
```

Zum anderen gibt es so genannte Blockkommentare. Diese werden mit `#comments-start` begonnen und mit `#comments-end` abgeschlossen. Der Text dazwischen wird gänzlich ignoriert. Alternativ sind auch die Kurzformen `#cs` und `#ce` möglich, Mischungen sind jedoch nicht erlaubt.

Listing 4.2: Mehrzeilige Kommentare

```
1 #comments-start
2 Das ist ein Beispielskript aus dem Buch
3 "AutoIt leicht gemacht".
4 #comments-end
5 Sleep(1000)
6 #cs
7 Der Befehl Sleep(1000) pausiert
8 das Skript 1000 Millisekunden lang,
```

```
9  also eine Sekunde.
10 #ce
```

4.3 Funktionen aufrufen

Das wichtigste Sprachelement in AutoIt sind die eingebauten Funktionen, die Ihre Skripte nutzen können und durch deren geschickte Aneinanderreihung das Skript seine besondere Aufgabe erfüllt.

Die Konventionen für den Aufruf einer solchen Funktion (und übrigens auch der UDFs [ref]) ist folgende:

Listing 4.3: Funktionsaufrufkonventionen

```
1 funktionsname(parameter_1, parameter_2, [ByRef] parameter_n )
```

Welche Parameter eine bestimmte Funktion entgegen nimmt, zeigt die Syntaxdarstellung in der Hilfe. Als Beispiel sei die Funktion **`ShellExecute`** betrachtet:

Listing 4.4: Syntaxbeispiel

```
1 ShellExecute ( "filename" [, "parameters" [, "workingdir" [, "verb" [,
2 showflag]]]] )
```

Diese Funktion kann also die Parameter `filename`, `parameters`, `workingdir`, `verb` und `showflag` entgegen nehmen.

Ist ein Parameter in dieser Syntax-Darstellung in eckige Klammern eingefasst, so ist er optional. Das heißt, die Funktion hat einen internen Standardwert für diesen Parameter. Falls Sie nichts anderes angeben, wird dieser Wert verwendet. Das bedeutet, dass dieser Aufruf völlig legitim ist:

Listing 4.5: Optionale Parameter

```
1 ShellExecute("C:\Windows\system32\notepad.exe")
```

Doch was nun, wenn Sie den Parameter `workingdir` angeben wollen, aber nicht `parameters`?

AutoIt kann nicht erkennen, zu welchem Parameter eine Angabe gehören soll. Die Verbindung des Aufrufes mit den Parametern kann nur über deren festgelegte Position gelingen:

Listing 4.6: Optionale Parameter mit Default

```
1 ShellExecute("C:\Windows\system32\notepad.exe", Default, "C:\")
```

Das Schlüsselwort `Default` ermöglicht es – obwohl drei Parameter angegeben werden müssen – für den zweiten Parameter den Standardwert zu verwenden.

Wenn ein Parameter in der Syntaxdarstellung mit dem Schlüsselwort `ByRef` gekennzeichnet ist, dann bedeutet das, dass sein Wert innerhalb der Funktion verändert werden kann und der letzte Stand wieder in der Ursprungsvariable verfügbar ist.

4.4 Variablen

Ein Grundpfeiler fast jeder Programmiersprache sind Variablen. Eine Variable kann man sich wie einen Container vorstellen, der einen Aufkleber auf der Außenseite trägt. Man kann AutoIt anweisen, einen Container bereitzustellen („Deklaration“), einen Wert darin abzulegen („Zuweisung“) oder den Inhalt des Containers zurückzugeben („Auslesen“). Für eine tiefer gehende Erklärung, warum dieses Beispiel auf AutoIt besonders gut passt, siehe [ref Variant].

4.4.1 Deklaration

Die Deklaration von Variablen ist in AutoIt recht einfach gelöst. Für eine Zeichenkette oder eine Zahl ist die allgemeine Syntax folgende:

Listing 4.7: Syntax Variablendeklaration

```
1 [Dim|Global|Local] $variablenname [= ("abc"|123)]
```

Variablen werden durch das voran gestellte Dollarzeichen ($) gekennzeichnet. Darauf folgt der Variablenname, der aus Groß- und Kleinbuchstaben, dem Unterstrich sowie Ziffern bestehen darf. Die Groß- und Kleinschreibung wird dabei nicht unterschieden, Sie sollten aber zur besseren Lesbarkeit versuchen, eine einheitliche Schreibweise zu wahren.

Nach einem = kann direkt bei der Deklaration ein Wert vorgegeben werden („Initialisierung“). Dabei sind Zeichenketten in einfache oder doppelte Anführungszeichen einzuschließen, Zahlen stehen ohne diese Einfassung.

Optional kann vor dem Variablennamen der Gültigkeitsbereich (engl. scope) der Variablen festgelegt werden. Eine globale Variable (`Global`) ist überall im Skript verfügbar

und kann ausgelesen werden. Variablen, die mit **Local** oder **Dim** definiert werden, gelten nur innerhalb der Funktion, in der sie deklariert wurden. Sind sie außerhalb von Funktionen deklariert, gelten sie überall (wie **Global**).

Es ist auch möglich, mehrere Variablen in einer Zeile zu deklarieren, indem man sie durch Kommata voneinander abtrennt.

Außerdem gibt es die Möglichkeit, Variablen mit aufeinanderfolgenden numerischen Werten mit dem Schlüsselwort **Enum** erzeugen zu lassen. Darauf soll aber hier nicht näher eingegangen werden.

Listing 4.8: Beispiele zur Deklaration von Variablen

```
1 Dim $alter, $beispiel
2 Dim $sprache = "Schwedisch"
3 Dim $anzahl = 3, $test
4 Local $counter
5 Local $text = "Finnisch"
6 Local $ausgabe = 2.56 ; Dezimalpunkt!
7 Global $temp
8 Global $autoit = "super"
9 Global $ausgabe = 3 ; Variable ist jetzt global
```

Das Weglassen der Angabe des Gültigkeitsbereiches wird wie **Dim** behandelt. Das eröffnet allerdings einige Fehlerquellen, sodass man mit dem Aufruf

Listing 4.9: Saubere Variablendeklaration erzwingen

```
1 Opt("MustDeclareVars", 1)
```

eine explizite Definition von Variablen (zumindest mit **Dim**) erzwingen kann.

4.4.2 Zuweisung

Werte weist man Variablen mit dem Operator = zu. Auf der linken Seite steht dabei der „Container“, rechts der neue Inhalt. Der Ausdruck auf der rechten Seite wird dabei solange ausgewertet, bis nur noch ein trivialer Wert (eine Zahl, eine Zeichenkette oder ein boole'scher Wert) übrig bleibt. Dieser Trivialwert wird dann in die Variable auf der linken Seite geschrieben. Diese Zuweisung kann entweder wie im vorherigen Abschnitt gezeigt direkt bei der Deklaration erfolgen, oder aber zu jedem späteren Zeitpunkt.

Listing 4.10: Zuweisungen an Variablen

```
1 Dim $alter = 23  ; 23
2 Global $text
```

```
$alter = 42       ; 42
$alter = 23 + 5   ; 28
$text = "Beispieltexte sind unlustig."
```

4.4.3 Auslesen

AutoIt akzeptiert an jeder Stelle, an der Zeichenketten oder Zahlen gefordert werden, auch Variablen und sonstige Ausdrücke. Die einzige Ausnahme ist der erste Parameter der Funktion „FileInstall“ [ref].

Der große Vorteil der Variablen in AutoIt ist, dass der Interpreter sich selbst darum kümmert, die Variable immer in den Datentyp umzuwandeln, der gerade gebraucht wird. Man muss sich also nicht wie in anderen Sprachen darum kümmern, immer die richtigen Typen zu konvertieren oder zu casten. Das bringt aber auch die Verantwortung mit sich, selbst darauf zu achten, was man in den Variablen speichert und ob das überhaupt wie gewünscht umwandelbar ist.

Beim Auswerten eines Ausdrucks wird der Variablenname von links nach rechts durch dessen Wert ersetzt.

Listing 4.11: Auslesen von Variablen

```
Global $zahl = 42
Global $text = "Antwort:"
MsgBox(0, "Antwort", 42)
MsgBox(0, $text, $alter)
```

4.4.4 Arbeiten mit Variablen

Neben den Stringfunktionen [ref] ist vor allem der Operator & bei der Arbeit mit Zeichenketten essentiell. Er verknüpft zwei Zeichenketten miteinander. Alle Teile werden dazu in Zeichenketten umgewandelt. Sehr hilfreich für das Verketten sind die Makros @TAB und @CRLF, die einen Tabulator oder einen Zeilenumbruch einfügen.

Listing 4.12: Operator &

```
Global $zahl = 42
Global $text = "Antwort:"
MsgBox(0, "Zahl?", $text & " " & $zahl)
```

4.5 Konstanten

Um im Bild mit den Variablen als Container zu bleiben, kann man sich Konstanten als durchsichtig versiegelte Container vorstellen. Ihnen wird einmalig ein Wert zugewiesen, der danach zwar noch ausgelesen, aber nicht mehr geändert werden kann. Die Deklaration erfolgt analog zu der von Variablen [ref], jedoch wird das Schlüsselwort **Const** vor dem Konstantennamen eingefügt.

Listing 4.13: Deklaration von Konstanten

```
1 Global Const $pi = 3.1415
2 Const $copyright = "Dieses Werk ist geschuetzt."
3 MsgBox(0, $pi, $copyright)
```

Der Zugriff auf den Wert einer Konstanten ist vollständig analog zum Zugriff auf eine Variable.

4.6 Makros

Makros lassen sich vielleicht am besten als *vorbelegte Konstanten* erklären: Ihr Wert ist wie bei Konstanten nicht änderbar (Ausnahme sind @error [ref] und @extended). Im Gegensatz zu normalen Konstanten müssen (und können) sie aber nicht deklariert und initialisiert werden. Beim Start eines Skriptes werden Sie von AutoIt mit den entsprechenden Werten belegt und stehen ab diesem Zeitpunkt zur Verfügung. Ihrer Besonderheit wegen haben Sie statt des $-Zeichens ein @-Zeichen am Anfang. So sind sie leicht von Variablen und Konstanten zu unterscheiden.

Einige wichtige Makros:

Makro	Inhalt
`@ScriptDir`	Verzeichnis, in dem das aktuelle Skript liegt
`@ScriptFullPath`	Kompletter Pfad zum Skript inklusive des Dateinamens
`@AutoItVersion`	AutoIt-Version
`@ComputerName`	Name des Computers im Netzwerk
`@UserName`	Name des angemeldeten Benutzers
`@IPAddress1`	IP-Adresse des ersten Netzwerkadapters
`@KBLayout`	Tastaturlayout
`@MyDocumentsDir`	Pfad zu den „Eigenen Dateien“
`@OSVersion`	Betriebssytemversion, z.B. WIN_XP oder WIN_VISTA
`@OSLang`	Systemsprache
`@ProgramFilesDir`	Programmverzeichnis, in der Regel `C:\Programme`

`@TempDir`	Verzeichnis für temporäre Dateien
`@Compiled`	1, falls das Skript kompiliert ist, ansonsten 0

Tabelle 4.2: Wichtige Makros

Eine vollständige Liste findet sich in der Hilfe.

Pfade sind stets ohne abschließenden Schrägstrich (\) hinterlegt. Die Pfadmakros sind in jedem Fall statischen Pfadangaben vorzuziehen, denn sie stellen sicher, dass Daten auch auf Computern, die nicht in den Standardpfaden installiert wurden, richtig abgelegt werden.

Listing 4.14: Verwendung von Makros

```
1 MsgBox(0, "Skriptverzeichnis", "Dieses Skript liegt im ↩
    Verzeichnis " & @CRLF &
2 @ScriptDir, 4)
3 ToolTip("Sie nutzen " & @OSVersion)
4 TrayTip("Programme werden installiert nach...", ↩
    @ProgramFilesDir, 20)
5 Sleep(3000)
```

Andere Makros beziehen sich auf bestimmte Bereiche von AutoIt, so @CRLF für Zeichenketten [ref] oder @HOUR, @MIN und @SEC für Zeitberechnungen. Sie werden im jeweiligen Abschnitt eingeführt.

4.7 Arrays

Arrays sind ein weiteres Konzept, das modernere Sprachen kennzeichnet. Es ermöglicht die organisierte Speicherung gleichartiger Daten unter einem gemeinsamen Namen.

Wichtig bei der Arbeit mit Arrays ist, dass Sie immer im Hinterkopf behalten, dass Arrays in AutoIt – wie in den meisten anderen Programmiersprachen auch – *nullbasiert* sind. Das bedeutet, dass der erste Wert den Index 0 bekommt, der zweite den Wert 1 und so weiter. Ein Array mit n Einträgen endet mit dem Eintrag $n-1$.

Um die Anzahl der Elemente eines Arrays herauszufinden, gibt es die Funktion `UBound` ↩ . Vor allem, wenn ein Array komplett durchlaufen werden soll, lässt sich damit leicht der Endwert beispielsweise einer For-Schleife bestimmen.

4.7.1 Eindimensionale Arrays

Die einfachste Form von Arrays sind eindimensionale Arrays. Sie ermöglichen das Speichern von verschiedenen Elementen, die sich über einen Index und den gemeinsamen Array-Namen ansprechen lassen.

4.7.1.1 Deklaration

Auch die Deklaration von eindimensionalen Arrays ist ähnlich zu der von normalen Variablen [ref]. Man gibt einfach nach dem Namen für das Array die Anzahl seiner Elemente in eckigen Klammern an. Auch die direkte Initialisierung kann bei Arrays erfolgen, dann ebenfalls in eckigen Klammern und die einzelnen Werte mit Kommata abgetrennt.

Listing 4.15: Deklaration von eindimensionalen Arrays

```
1 Const $anzahl_autobahnen = 93
2 Global $autobahnen[$anzahl_autobahnen]
3 Dim $swe_num[5] = ["noll", "ett", "tva", "tre", "fyra"]
4 $copyright[4] = ["Dieses", "Werk", "ist", "geschuetzt."]
```

Im Nachhinein kann die Größe des Arrays mit dem Befehl **ReDim** verändert werden.

4.7.1.2 Zuweisung

Die Zuweisung von Werten an die einzelnen Elemente erfolgt über den Namen des Arrays und den in Klammern angegeben Index. Letzterer wird wieder array-typisch in eckige Klammern gesetzt.

Listing 4.16: Zuweisungen an eindimensionale Arrays

```
1 Global $fruechte[4]
2 $fruechte[0] = "Apfel"
3 $fruechte[2] = "Birne"
4 MsgBox(0, "Frucht 3", $fruechte[2])
```

4.7.1.3 Auslesen

Das Lesen aus einem Array funktioniert ebenso über den Namen sowie den Index.

Listing 4.17: Zugriff auf eindimensionale Arrays

```
1 Global $fruechte[4]
```

[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]	[0][6]	[0][7]
[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]	[1][6]	[1][7]
[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]	[2][6]	[2][7]
[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]	[3][6]	[3][7]
[4][0]	[4][1]	[4][2]	[4][3]	[4][4]	[4][5]	[4][6]	[4][7]
[5][0]	[5][1]	[5][2]	[5][3]	[5][4]	[5][5]	[5][6]	[5][7]
[6][0]	[6][1]	[6][2]	[6][3]	[6][4]	[6][5]	[6][6]	[6][7]
[7][0]	[7][1]	[7][2]	[7][3]	[7][4]	[7][5]	[7][6]	[7][7]

Abbildung 4.1: Darstellung eines Schachbrettes in einem 2-dimensionalen Array

```
2 $fruechte[0] = "Apfel"
3 $fruechte[2] = "Birne"
4 MsgBox(0, $fruechte[0], $fruechte[2])
```

4.7.2 Mehrdimensionale Arrays

Noch mehr Möglichkeiten, reale Sachverhalte in AutoIt abzubilden, erschließen sich mit mehrdimensionalen Arrays. Stellen Sie sich vor, ein Schachbrett sollte repräsentiert werden. Mit einem eindimensionalen Array aus 64 Elementen ist das nicht wirklich intuitiv. Anders sieht es aus, wenn Sie dafür eine zweidimensionale Struktur verwenden, die man sich als Tabelle vorstellen kann [ref Abb].

4.7.2.1 Deklaration

Die Deklaration eines mehrdimensionalen Arrays funktioniert analog zu der eindimensionaler Arrays. Es sind dabei bis zu 64 Dimensionen möglich.

Listing 4.18: Deklaration von mehrdimensionalen Arrays

```
1 Global $schachbrett[8][8]
2 Dim $tabelle[3][2] = [["A", "a"], ["B", "b"], ["C","c"]]
```

4.7.2.2 Zuweisung

Auch die Zuweisung von Werten erfolgt in ähnlicher Weise wie bei den Arrays mit nur einer Dimension:

Listing 4.19: Zuweisung von mehrdimensionalen Arrays

```
1 Dim $tabelle[3][2]
2 $tabelle[0][1] = "Spalte 1"
3 $tabelle[1][1] = "Wert 1"
```

4.7.2.3 Auslesen

Zu guter Letzt können Sie Werte aus mehrdimensionalen Arrays auch auslesen wie Sie es bereits von den eindimensionalen Arrays kennen.

Listing 4.20: Auslesen von mehrdimensionalen Arrays

```
1 Dim $tabelle[3][2] = [["A", "a"], ["B", "b"], ["C","c"]]
2 MsgBox(0, "", $tabelle[1][0])
```

4.7.3 Array.au3

Die UDF Array.au3 enthält eine Menge Funktionen für den erleichterten Umgang mit Arrays. Einige interessante davon:

Funktionsname	Beschreibung
_ArrayAdd	Fügt einen Wert an das Ende eines Arrays an
_ArrayDelete	Löscht einen Wert aus einem Array
_ArrayDisplay	Zeigt alle Werte eines Arrays übersichtlich an
_ArrayInsert	Fügt einen Wert an einer gegebenen Stelle in ein Array an
_ArrayMax	Liefert den Maximalwert aus einem Array von Zahlen
_ArrayMin	Liefert den Minimalwert aus einem Array von Zahlen
_ArrayPop	Löscht das letzte Element eines Arrays
_ArrayPush	Fügt einen Wert ein und löscht einen anderen
_ArrayReverse	Dreht die Werte eines Arrays um
_ArraySearch	Sucht in einem Array nach einem Wert
_ArraySort	Sortiert ein Array
_ArrayToString	Fügt ein Array zu einem String zusammen

Tabelle 4.4: Funktionen aus der Array.au3

4.8 Bedingungen

Sobald ein Skript eine gewisse Komplexität erreicht, wird es unvermeidlich, eine Möglichkeit zu schaffen, um zur Laufzeit flexibel reagieren zu können.

4.8.1 If-Abfragen

If-Then-Konstrukte sind von zentraler Bedeutung, um abhängig von einem Wert verschiedene Reaktionen auslösen zu können.

Die Syntax lautet wie folgt:

Listing 4.21: If-Then-Syntax

```
If Ausdruck Then
    [Reaktion]
[ElseIf Ausdruck Then
     [Reaktion]]
[Else
     [Reaktion]]
EndIf
```

Ein `Ausdruck` kann in AutoIt sehr vielfältig gestaltet sein. Der einfachste Fall ist ein expliziter Vergleich mit dem *Vergleichsoperator* = (nicht zu verwechseln mit dem *Zuweisungsoperator* =). Sind beide Seiten des Operators nach der Auswertung gleich, so ist der Ausdruck wahr und es wird der Teil nach dem **`Then`** ausgeführt. Andernfalls wird beim nächsten **`Else`**, **`ElseIf`** oder **`EndIf`** fortgefahren. Funktionsaufrufe sind auf beiden Seiten möglich. Sie werden ausgeführt und dann durch ihren Rückgabewert ersetzt, dessen Wert sich in der Hilfe ablesen lässt.

Neben = unterstützt AutoIt noch andere Vergleichsoperatoren, namentlich < (kleiner), <= (kleiner oder gleich), > (größer), >= (größer oder gleich) und <> (ungleich).

Listing 4.22: Beispiele If-Anweisungen

```
Global $zahl = 42 ; Variieren Sie diesen Wert
                  ; und beobachten Sie die
                  ; Auswirkungen.
If $zahl = 31 Then
  MsgBox(0, "", "Die Zahl ist 31!")
Else
  MsgBox(0, "", "Die Zahl ist nicht 31!")
EndIf
```

```
Global $zahl2 = 23 ; Variieren Sie diesen Wert
                   ; und beobachten Sie die
                   ; Auswirkungen.
If $zahl < 30 Then
  MsgBox(0, "", "Die Zahl ist kleiner als 30!")
ElseIf $zahl > 30 Then
  MsgBox(0, "", "Die Zahl ist groesser als 30!")
Else
  MsgBox(0, "", "Die Zahl ist 30!")
EndIf
```

Lässt man den Vergleichsoperator weg, so wird intern `<> 0` angehängt. Das bedeutet, dass man viele Funktionen, die als Rückgabewert bei Erfolg 1 und bei Misserfolg 0 haben, folgendermaßen einfach testen kann:

Listing 4.23: Erfolgskontrolle mit If-Anweisungen

```
If FileExists("C:\autoexec.bat") Then
    MsgBox(0, "", "C:\autoexec.bat existiert.")
Else
    MsgBox(0, "", "C:\autoexec.bat existiert nicht.")
EndIf
```

AutoIt unterstützt die *Boole'schen Operatoren* **and**, **or** und **not**. Damit kann man komplexe Ausdrücke darstellen, die von mehreren Parametern abhängen. Es kann dabei erforderlich sein, Klammern zu setzen, um die Auswertungsreihenfolge durch den Interpreter zu bestimmen – schaden können Klammern nie; die Übersicht dagegen fördern sie oft.

Listing 4.24: Boole'sche Operatoren

```
If FileExists("C:\autoexec.bat") and FileGetSize("C:\boot.
   ini") < 1024 Then
    MsgBox(0, "", "Hier.")
Else
    MsgBox(0, "", "Dort.")
EndIf

If (not 1<>9) or (2>-2 and 3*2=6) Then
    MsgBox(0, "", "Hier.")
Else
    MsgBox(0, "", "Dort.")
EndIf
```

Boole'sche Operatoren
Die sogenannten „Boole'schen Operatoren" gehen auf den englischen Mathematiker George Boole (1815 – 1864) zurück. Sie sind Verknüpfungen zwischen Aussagen. Eine Aussage ist in diesem Zusammenhang entweder wahr (1) oder falsch (0). Andere Zustände gibt es nicht.

Alle Codeteile, die einen Vergleichsoperator enthalten, bilden eine Aussage. `5>6` ist beispielsweise falsch, `"test"="test"` dagegen wahr.

Die einfachste Operation, die man auf eine der Aussagen anwenden kann, ist die Negation **NOT**. Aus einer wahren Aussage wird dabei eine falsche (1 ▷ 0) und aus einer falschen eine wahre (0 ▷ 1).

Als nächstes können Sie **OR**-Verknüpfungen herstellen. In AutoIt ist dieses OR ein „Logisches Oder". Das bedeutet, dass nur eine der Aussagen wahr (true) sein muss, damit die Gesamtaussage wahr ist. Nur für den Fall dass beide Aussagen falsch sind, ist auch die Verknüpfung falsch.

Die dritte in AutoIt verfügbare Boole'sche Operation ist **AND**. Diese ist quasi das Gegenteil der Or-Verknüpfung: Nur wenn beide Aussagen wahr sind, ist auch das Ergebnis wahr, andernfalls ist es falsch.

Ausdrücke, die mehr als zwei Aussagen enthalten, werden von links nach rechts ausgeführt. Das Ergebnis einer Verarbeitung von je zwei Aussagen wird eingesetzt und dann erneut die Prüfung gestartet. Man kann dieses Verhalten wie in der Mathematik mit runden Klammern steuern.

[todo Beispiele]

Hängt nur ein Befehl von der If-Abfrage ab, so kann man die If-Abfrage verkürzen:

Listing 4.25: Kurzschreibweise für If-Anweisung

```
1 If 1=1 Then MsgBox(0, "", "1 ist 1")
2 MsgBox(0, "", "Diese Box erscheint immer.")
```

4.8.2 Select

Für Fälle, in denen man eine große Zahl von Ausdrücken testen und darauf reagieren möchte, wird die Struktur mit If-Anweisungen schnell unhandlich. Als Alternative bietet sich eine Select-Konstruktion an. Deren Syntax stellt sich wie folgt dar:

Listing 4.26: Syntax von Select

```
1 Select
```

```
  Case Ausdruck
    [Reaktion]
  Case Ausdruck
    [Reaktion]
  Case Else
    [Reaktion]
EndSelect
```

Ist der Ausdruck hinter dem ersten **Case** wahr, so werden alle Befehle bis zum nächsten **Case** ausgeführt und danach nach **EndSelect** fortgefahren. Die folgenden **Case**s werden also nicht mehr ausgeführt, selbst wenn der Ausdruck dahinter wahr sein sollte.

Ist der Ausdruck hinter dem ersten **Case** nicht wahr, so wird der nächste **Case** abgearbeitet. Analog wird für alle anderen **Case**-Abschnitte verfahren. Der Abschnitt **Case Else** wird in jedem Fall ausgeführt, wenn die Prüfung bis an diese Stelle kommt, also vorher kein Ausdruck `wahr` ergeben hat.

Das Verhalten, dass nach der Reaktion auf einen wahren Ausdruck die restlichen **Case**s übersprungen werden, kann man verhindern, indem man das Schlüsselwort **ContinueCase** am Ende des zuerst ausgeführten Abschnittes einfügt.

4.8.3 Switch

Die **Switch**-Anweisung arbeitet ähnlich wie **Select**. Der wesentliche Unterschied ist, dass hier eine Variable (oder Konstante) angegeben wird, deren Wert in allen folgenden **Case**s untersucht wird.

Listing 4.27: Syntax von Switch

```
Switch Variable
  Case Ausdruck
    [Reaktion]
  Case Ausdruck
    [Reaktion]
  Case Else
    [Reaktion]
EndSwitch
```

Mittels des Schlüsselwortes **to** ist es möglich, ganze Zahlenbereiche gemeinsam zu verarbeiten. Da die grundlegende Verarbeitungsreihenfolge bei **Switch** und **Select** gleich ist, hat auch das Schlüsselwort **ContinueCase** hier den gleichen Effekt.

Listing 4.28: Beispiel zu Switch

```
Global $zahl = 48/2  ; 24
Switch $zahl
  Case 1
    MsgBox(0, "", "$zahl = 1")
  Case 14 to 30
        MsgBox(0, "", "$zahl von 14 bis 30")
  Case Else
        MsgBox(0, "", "$zahl nicht erfasst")
EndSwitch
```

4.9 Schleifen

Neben der bedingten Ausführung von Quelltext, die mit den im vorigen Abschnitt dargestellten Methoden in AutoIt durchgeführt werden kann, zählen Schleifen zu den grundlegenden Eigenschaften einer Praogrammiersprache, die sinnvolles Programmieren erlaubt.

In AutoIt gibt es drei Typen von Schleifen, die die wichtigsten Fälle in der Praxis abzubilden vermögen.

4.9.1 For . . . Next

Eine For-Schleife bietet sich immer dann an, wenn man vor dem Beginn der Schleife sagen kann, wie oft diese durchlaufen werden soll. Dies gilt auch, wenn das abhängig vom Wert einer Variablen ist.

Listing 4.29: Beispiel zu For . . . Next-Schleifen

```
For $i = 1 to 10 step 1
   MsgBox(0, "", "Durchlauf: " & $i)
Next
```

Es wird eine Zählvariable deklariert (hier `$i`), die einen Startwert (1) und Endwert (10) bekommt. Die Angabe der Größe der Schritte, in denen die Zählvariable verändert wird, kann hinter dem Schlüsselwort **`step`** erfolgen, ist aber optional.

Die in der ersten Zeile initialisierte Variable steht innerhalb der Schleife zur Verfügung und kann wie jede andere Variable angesprochen und ausgelesen werden. Auch Schreibzugriff ist möglich, allerdings mit Vorsicht zu genießen, da damit die Anzahl der Schleifendurchläufe verändert wird.

Mit einem negativen Wert für den **step** kann eine rückwärts zählende Schleife erzeugt werden.

Traditionell werden Zählvariablen beginnend mit `$i` dem Alphabet folgend mit einzelnen Buchstaben benannt. Das ist aber selbstverständlich nicht bindend.

4.9.1.1 For . . . In . . . Next

Eine Abwandlung der For-Schleife ist die For . . . In-Schleife. Damit kann einfacher ein komplettes Array oder eine Collection von Objekten durchlaufen werden. Dabei wird eine zusätzliche Variable angegeben, in die der Wert des jeweiligen Durchlaufes kopiert wird.

Listing 4.30: Beispiel zu For . . . In . . . Next-Schleifen

```
1 Dim $aArray[4] = ["a", "b", "c", "d"]
2
3 For $element In $aArray
4     MsgBox(0, "", $element)
5 Next
```

4.9.2 Do . . . Until

Wenn im Gegensatz zur For . . . Next-Schleife nicht im Vorhinein klar ist, wie viele Durchläufe benötigt werden, dann bietet sich beispielsweise eine Do . . . Until-Schleife an.

Dabei wird eine Abbruchbedingung definiert, die nach jedem Durchlauf geprüft wird. Ist der Ausdruck wahr, so wird die Schleife beendet, andernfalls erfolgt der nächste Durchlauf ab **Do**.

Listing 4.31: Beispiel zu Do . . . Until-Schleifen

```
1 Global $i = 4
2 Do
3   $i = $i + $i
4 Until $i > 90
5 MsgBox(0, "Result", $i)
```

Die Prüfung am Ende eines Durchlaufes bedeutet, dass der Anweisungsblock innerhalb der Schleife mindestens einmal ausgeführt wird.

Mit dem Schlüsselwort **ExitLoop** kann die Schleife verlassen werden, nach **ContinueLoop** wird unabhängig von der Position innerhalb der Schleife sofort der nächste Durchlauf begonnen.

4.9.3 While . . . WEnd

Die While . . . WEnd- ist der Do . . . Until-Schleife insofern ähnlich, dass es keine vor Beginn vorgegebene Zahl von Durchläufen gibt.

Es gibt statt dessen eine Fortsetzungsbedingung, deren Angabe im Schleifenkopf erfolgt, sodass es sein kann, dass die Schleife kein einziges Mal durchlaufen wird.

Listing 4.32: Beispiel zu While . . . WEnd-Schleifen

```
1 Global $i = 4
2 While $i < 100
3   $i = $i + 5
4 WEnd
5 MsgBox(0, "Result", $i)
```

Wie bei der Do . . . Until-Schleife gilt auch hier, dass mit dem Schlüsselwort **ExitLoop** die Schleife verlassen werden kann. Nach **continueLoop** wird unabhängig von der Position innerhalb der Schleife sofort der nächste Durchlauf begonnen.

Mit diesem Schleifentyp ist es auch ein Leichtes, eine Endlosschleife zu bauen:

Listing 4.33: Endlosschleife

```
1 While 1=1
2   ; hier stehen die Befehle,
3   ; die immer wieder ausgefuehrt
4   ; werden sollen
5 WEnd
```

5 Fortgeschrittene Themen

Nachdem nun die theoretischen Grundlagen gelegt sind – die nebenbei bemerkt nicht nur für AutoIt gelten, sondern für die meisten Programmiersprachen in ähnlicher Weise – liegt im folgenden Kapitel das Augenmerk auf praktischen Funktionen von AutoIt. Diese werden blockweise vorgestellt und mit Beispielen eingeführt.

5.1 User Defined Functions

Sowohl im Lieferumfang von AutoIt als auch im Internet finden sich zahlreiche so genannte UDFs. Diese Abkürzung steht für „User Defined Functions“, also Funktionen, die von Nutzerseite geschrieben werden (können). Sie verwenden die bereits in AutoIt eingebauten Funktionen und kombinieren sie so miteinander, dass neue Möglichkeiten erschlossen werden.

Im Allgemeinen wird der Begriff *UDF* dann für eine Sammlung mehrerer Funktionen aus dem gleichen Funktionskreis verwendet, die in einer Datei zusammengefasst werden. UDFs sind damit von ihrer Stellung her den Bibliotheken anderer Programmiersprachen (C/C++, Java, ...) vergleichbar.

Um die Funktionen einer UDF nutzen zu können, bindet man sie mit der Compiler-Direktive ein, z. B. `#include <Array.au3>`. Vor dem Ausführen des – kompilierten oder unkompilierten – Skriptes wird an der Stelle, an dem das **Include** steht, die referenzierte Datei eingebunden. Man kann auf diesem Weg also auch sein Skript in mehrere Dateien splitten, die aber dennoch wie ein Skript reagieren.

Bei UDFs, die Funktionen bereitstellen, ist darauf zu achten, dass sie nicht innerhalb eigener Funktionen ([ref] Func usw.), also zwischen **Func** und **EndFunc** eingebunden werden, da AutoIt verschachtelte Funktionen nicht unterstützt.

Ist eine UDF eingebunden, kann man auf ihre Funktionen wie auf die internen Funktionen von AutoIt auch über ihren Namen zugreifen und Parameter analog übergeben.

Bei der **Include**-Syntax ist wichtig, dass die entsprechende Datei im Verzeichnis „Include“ gesucht wird, wenn der Dateiname in < und > eingeschlossen ist. Bei einer Standardinstallation der stabilen Version ist dies der Pfad `C:\Programme\AutoIt v3\Include`, bei der Beta-Version `C:\Programme\AutoIt v3\Beta\Include` für die Betaversion. Bei abweichenden Installationsverzeichnissen lauten die Pfade entspre-

chend anders. Man kann den Ort der UDF auch relativ zum Skriptverzeichnis angeben, indem man ihren Namen mit“ und “ ummantelt. In beiden Fällen ist es möglich, relative Pfade anzugeben, also beispielsweise in Unterordnern suchen zu lassen.

Listing 5.1: Include

```
#include <Array.au3>
#include <Database\SQLite.au3>
#include "my_funcs.au3"
#include "my_inlcudes\my_funcs.au3"
$test = _ArrayCreate("1", "2", "3")
```

Das obige Beispiel kann nur ausgeführt werden, wenn die entsprechenden Dateien existieren. Andernfalls bricht AutoIt mit einer Fehlermeldung ab.

5.2 Programme ausführen

Eine Grundfunktion, die AutoIt schon seit den frühsten Versionen bietet, ist das Ausführen von externen Programmen. Früher gab es dafür den Befehl **Run**. Mit Version 3.2.2.0 [?] wurde jedoch ein weit mächtigeres Werkzeug in die Hand der Skripter gelegt: **ShellExecute** . Diese Funktion bezieht ihre Stärke daraus, dass Sie direkt auf die gleichnamige Windows-API-Funktion gemappt ist, also auf die gleichen Fähigkeiten zurückgreifen kann, die beispielsweise auch der Windows Explorer nutzt.

Im Gegensatz zum Befehl **Run** funktioniert nun auch das Starten von Programmen in Pfaden, die in der Umgebungsvariablen `PATH` eingetragen sind, reibungslos.

Neben dem Dateinamen mit Dateipfad als erstem Parameter kann man **ShellExecute** Parameter für das zu startende Programm mitgeben. Das Arbeitsverzeichnis kann danach noch angegeben werden, was aber nur in Sonderfällen nötig ist. Ein sehr mächtiger ist der vierte Parameter namens „verb“. Hiermit kann man die für den jeweiligen Dateityp eingetragenen Aktionen direkt auslösen. Für einige Fälle kann so direkt mit **ShellExecute** gedruckt werden. Als letzter Parameter steuert „showflag“, in welcher Ansicht das Programm geöffnet werden soll.

Listing 5.2: ShellExecute

```
ShellExecute("notepad")
; eine Eingabeaufforderung maximiert starten
ShellExecute("cmd.exe", "", "C:\Windows", "open", @SW_MAXIMIZE)
; ein Bild mit dem Standardprogamm oeffnen
ShellExecute("C:\Windows\Blaue Spitzen 16.bmp")
```

```
6 ; die Boot.ini direkt drucken:
7 ShellExecute("C:\boot.ini", "", "", "print")
```

Wenn es nötig ist, das Skript erst fortzusetzen, nachdem das gestartete Programm beendet ist, dann bietet sich `ShellExecuteWait` an. Dieser Befehl hat die gleichen Parameter wie **`ShellExecute`**, wartet aber wie gewünscht.

DOS-Befehle bieten oft eine schnelle Lösung für spezielle Probelmstellungen in der Verwaltung des Computers. Diese Kommandos lassen sich direkt von AutoIt auslösen. Dazu verwenden Sie den Befehl `_RunDOS`. Dazu muss die UDF „Process.au3“ eingebunden werden.

5.3 Maus- und Tastatursimulation

Ein weiteres Feature, das AutoIt schon seit Beginn der Entwicklung auszeichnet, und das die Sprache für den Alltag besonders interessant macht, ist die Möglichkeit, alle Aktionen, die per Hand mit der Maus oder der Tastatur ausgeführt werden, zu automatisieren.

5.3.1 Mausfunktionen

Für die Simulation von Mausaktionen stehen die folgenden Funktionen zur Verfügung:

Funktionsname	Beschreibung
`MouseMove`	bewegt die Maus an die angegebene Position
`MouseClick`	Führt einen oder mehrere Mausklicks aus
`MouseDown`	Drückt eine Maustaste nach unten und hält sie gedrückt
`MouseUp`	Lässt eine Maustaste wieder los
`MouseClickDrag`	Führt eine Drag&Drop-Aktion aus
`MouseWheel`	Simuliert ein Drehen des Mausrades

Tabelle 5.2: Mausfunktionen

Die jeweiligen Parameter entnehmen Sie bitte der aktuellen Hilfe-Datei.

Ein nettes Gimmick ist die natürlich nachempfundene Bewegung der Maus, die vor allem bei dem Befehl **`MouseMove`** deutlich wird. Damit lässt sich AutoIt auch für ansprechende Demonstrationen von anderen Programmen oder deren Funktionen einsetzen.

Mit folgenden **Opt**s lassen sich grundlegende Einstellungen für diese Mausfunktionen vornehmen: `MouseClickDelay`, `MouseClickDownDelay`, `MouseClickDragDelay` und `MouseCoordMode`

Mit der Option `MouseCoordMode` lässt sich regeln, wie AutoIt die Angaben der x- und y-Koordinaten interpretiert. Dabei sind folgende Werte möglich:

0 Die Koordinaten werden relativ zum aktiven Fenster interpretiert

1 Die Koordinaten werden relativ zum gesamten Desktop aufgefasst

2 Die Koordinaten werden relativ zum verwendbaren Teil (client area) des aktiven Fensters verwendet

Bei dem Wert 2 werden die Titelleiste und die Ränder im Gegensatz zu Status 0 nicht mit gezählt.

Der Vorteil dieser beiden Werte ist, dass Klicks damit nicht mehr davon abhängig sind, an welcher Stelle auf dem Desktop das Fenster erscheint. Allerdings können unterschiedliche Bildschirmauflösungen immer noch Probleme verursachen.

Entsprechende gelten die Angaben für die Funktion PixelCoordMode [ref].

Listing 5.3: Mausfunktionen

```
1 MouseMove(1, 1)
2 MouseClick("left", 200, 200, 2)
3 MouseWheel("down", 3)
```

Wie Sie vielleicht beim Ausführen der Beispiele bemerkt haben, kann diese Steuerung zu Problemen führen. Denn unter anderen Umständen (andere Fenster im Vordergrund, Fenster verschoben, unterschiedliche Bildschirmauflösung) können diese Mausaktionen unerwünschte Ergebnisse liefern. Daher sollten Sie diese Funktionen mit Bedacht einsetzen.

5.3.2 Tastaturfunktionen

Die wichtigste Funktion, die AutoIt mitbringt, um Tastendrücke zu simulieren, ist die **Send**-Funktion.

Mit dieser lässt sich nahezu alles, was ein Anwender an seiner Tastatur durchführen kann, automatisch ausführen. Die Tastendrücke kommen genau so an, als hätten sie real stattgefunden (siehe zu Ausnahmen [ref bots, spiele].

Obwohl **Send** in nur einem Parameter die Tastendrücke und -kombinationen entgegennimmt (für die Bedeutung des zweiten siehe [ref.faq.send.komisch]), ist die Syntax relativ komplex:

Um Sondertasten zu senden, gibt es die Zeichen ! (Alt-Taste), ^ (Strg-Taste), # (Windows-Taste) und + (Shift-Taste). Damit lassen sich einfach Tastenkombinationen senden:

Listing 5.4: Senden von Tastenkombinationen mit Sondertasten

```
1 ShellExecute("notepad")
2 WinWait("Unbenannt - Notepad")
3 WinActivate("Unbenannt - Notepad")
4 WinWaitActive("Unbenannt - Notepad")
5 Sleep(1000)
6 Send("!d") ; oeffnet das Menue "Datei"
7 Sleep(2000)
8 Send("b") ; Waehlt "Beenden"
```

Hier sind auch Kombinationen mit mehr als 2 Tasten möglich, beispielsweise **Send** ("!+d") für Alt+Shift+D.

Um weitere Spezialtasten simulieren zu können, gibt es eine entprechende Liste mit Codes in der Hilfedatei [ref]. Ein Auszug daraus:

Code	Taste
{SPACE}	Leertaste
{Enter}	Return / Enter
{BS}	Backspace
{DEL}	Entfernen
{UP}, {DOWN}, {LEFT}, {RIGHT}	Cursortasten (hoch, runter, links, rechts)
{ESC}	Escape
{F1} – {F12}	F1 - F12
{TAB}	Tabulator

Tabelle 5.4: Tastencodes für **Send**

Um einen Tastendruck mehrfach zu wiederholen, kann man die Anzahl in den geschweiften Klammern, nach einem Leerzeichen mit angeben:

Listing 5.5: Mehrfaches Senden von Tastendrücken

```
1 Send("!{TAB 4}") ; haelt Alt gedrueckt und sendet 4 mal Tab
```

Mit folgenden **Opt**ionen lässt sich das Verhalten der Funktion **Send** beeinflussen: `SendAttachmode`, `SendKeyDelay`, `SendKeyDownDelay` und `SendCapslockMode`.

Die Funktion **ControlSend** kann die Tastendrücke direkt an ein Control übergegen. Auf diese Weise können sie nicht verloren gehen, weil beispielsweise ein anderes Fenster aktiv ist. Dazu kommt damit die Möglichkeit, auch Fenster, die gerade im Hintergrund oder gar versteckt sind, zu erreichen.

5.3.3 HotKeys

[todo]

5.4 Aktionen mit fremden Anwendungen

AutoIt kann fremde Programme in verschiedener Weise fernsteuern. Das ist insbesondere für automatisierte Installationen [ref] hilfreich, aber auch in vielen anderen Szenarien kann es notwendig sein, mit Fremdprogrammen umzugehen.

5.4.1 Fenster

Mit einer Reihe von Funktionen ist es möglich, Fenster beeinflussen:

Funktion	Beschreibung
WinActivate	Fenster aktivieren
WinClose	Fenster „sanft“ schließen
WinKill	Fenster „hart“ schließen (erzwungen)
WinFlash	Fenster in der Taskleiste blinken lassen
WinMinimizeAll	Alle Fenster minimieren (wie Win + M)
WinMinimizeAllUndo	Fenster wiederherstellen nach WinMinimizeAll
WinMove	Fenster bewegen *und* Größe ändern
WinSetOnTop	Fenster dauerhaft in den Vordergrund holen
WinSetState	Fensterstatus ändern (verstecken, wieder anzeigen, minimieren, maximieren, wiederherstellen, deaktivieren, aktivieren)
WinSetTitle	Fenstertitel neu setzen

WinSetTrans	Fenstertransparenz setzen (nur Windows 2000/XP/Vista)

Tabelle 5.6: Fensterkontrollfunktionen

Hinweise:

- Die korrekten Werte für die Parameter title und text lassen sich mit dem AutoIt Window Info Tool herausfinden [ref]
- Wenn Sie den Parameter title als leeren String angeben (`""`), dann wird der Befehl auf das gerade aktive Fenster bezogen
- Die Art, wie der Titel erkannt wird, steuern Sie über die Option WinTitleMatchMode [ref]
- Der Parameter text wird immer im gesamten Fenstertext gesucht, entspräche also dem WinTitleMatchMode 2.

Dazu kommen einige Funktionen, die vornehmlich für automatische Installationen sehr nützlich sind:

Funktion	Beschreibung
WinActivate	Fenster aktivieren
WinWait	Wartet, bis das angegebene Fenster existiert
WinWaitActive	Wartet, bis das angegebene Fenster aktiv ist
WinWaitNotActive	Wartet, bis das angegebene Fenster nicht mehr aktiv ist
WinWaitClose	Wartet, bis das angegebene Fenster geschlossen ist

Tabelle 5.8: Fensterfunktionen für automatische Installationen

Wie oft AutoIt bei den `Wait`-Funktionen nachprüfen soll, ob der gewünschte Zustand erreicht ist, lässt sich mit der Option `WinWaitDelay` einstellen.

Der letzte Parameter eröffnet die Möglichkeit, die maximale Wartezeit zu begrenzen und so zu verhindern, dass das Skript an dieser Stelle hängen bleibt.

Mehr zur Anwendung dieser Funktionen finden Sie in [ref install].

Die Funktion **WinList** liefert eine Liste aller existierenden Fenster zurück. Damit können alle GUIs schnell durchlaufen und gemeinsam manipuliert werden.

Eine weitere Gruppe bilden die Funktionen, die AutoIt bietet um Eigenschaften und Zustände von Fenstern auszulesen. Sie sind die Grundlage, auf der man Reaktionen planen und durchführen kann:

Funktion	Beschreibung
WinActive	Hat das Fenster den Fokus?
WinExists	Existiert das Fenster?
WinGetClassList	Welche Controls befinden sich in der GUI?
WinGetHandle	Wie ist das Handle des Fensters?
WinGetPos	Wo befindet sich die GUI?
WinGetProcess	Welche ID hat der Prozess, zu dem das Fenster gehört?
WinGetState	Welchen Status hat das Fenster? (siehe auch **WinSetState**)
WinGetTitle	Wie lautet der Titel des Fensters?
WinGetText	Welcher Text befindet sich in dem Fenster?

Tabelle 5.10: Fenstereigenschaften auslesen

Über die Option WinTitleMatchMode lässt sich steuern, wann die Funktionen, die sich auf ein Fenster beziehen, ein solches erfassen.

Es gibt dabei folgende gültige Werte:

1. Der Titel muss so anfangen, wie angegeben
2. Der Titel muss den angegebenen Teil an einer beliebigen Stelle enthalten
3. Der Titel muss exakt übereinstimmen
4. Sondermodus: Bei der Eingabe von `classname=SysListView32` wird das Fenster mit dem entsprechenden Classname gefunden, bei `regex=^[a-z]$` wird aufgrund eines Regulären Ausdrucks [ref] gesucht.

Voreingestellt ist der Modus 1. Es ist möglich, den Wert jederzeit im Skript neu zu setzen. Falls mehrere Fenster die geforderten Kriterien erfüllen, wird die zuletzt aktive GUI angesprochen.

Wenn Sie der Zahl ein Minus ($-$, z.B. -2) voranstellen, wird zusätzlich die Groß-/Kleinschreibung ignoriert, Sie müssen aber den Titel in Kleinbuchstaben angeben. [!?]

Das folgende Beispiel kombiniert einige der genannten Befehle, die AutoIt bietet, um wie von Geisterhand mit fremden Fenstern zu arbeiten. Beim ersten Kontakt mit einem solchen Skript ist in aller Regel ein wirklich Aha-Effekt festzustellen:

Listing 5.6: Beispiel für Fensterbefehle

```
; Notepad starten
ShellExecute("notepad")
Opt("WinTitleMatchMode", 2)
; Auf Notepad warten
WinWait("Editor")
Sleep(4000)
; Fenster minimieren
WinSetState("Unbenannt", "", @SW_MINIMIZE)
Sleep(4000)
; Wenn das Fenster minimiert ist, dann maximieren
If BitAnd(WinGetState("Unbenannt - "), 16) Then WinSetState
   ("Editor", "",
@SW_MAXIMIZE)
; Das Fenster wieder aktivieren
WinActivate("Editor")
; Warten, bis das Aktivieren abgeschlossen ist
WinWaitActive("Editor")
Send("Hier spricht der Pilot.")
Sleep(4000)
; Fenster "sanft" schliessen
WinClose("Editor", "der Pilot")
; "Speichern?"-Abfrage abfangen
WinWait("Editor", "&Nicht speichern")
; Abfrage in den Vordergrund bringen
WinActivate("Editor", "&Nicht speichern")
; Warten, bis das geschehen ist
WinWaitActive("Editor", "&Nicht speichern")
; Jetzt landet der Send-Befehl sicher richtig
Send("{RIGHT}{SPACE}")
```

5.4.2 Controls

5.5 Daten vom Benutzer einlesen

[InputBox] [ref GUI]

5.6 Daten speichern

Häufig möchte man Ergebnisse von Berechnungen, Programmeinstellungen oder Daten anderer Art über die Skriptlaufzeit hinaus festhalten.

Dazu bietet AutoIt verschiedene integrierte Methoden, namentlich Ini-Dateien, die Windows Registry und allgemein Textdateien. Des Weiteren gibt es noch Datenbanken, auf die aber erst später eingegangen werden soll [ref].

5.6.1 Ini-Dateien

Noch aus frühen Zeiten von Windows stammen die Ini-Dateien, die dazu gedacht waren, Konfigurationseinstellungen von Programmen zu speichern.

Diese Dateien besitzen Kategorien („sections“), in denen es verschiedene Schlüssel („keys“) mit jeweils einem zugehörigen Wert („value“) geben kann.

Wichtig ist dabei, dass die sowohl „sections“ untereinander als auch die einzelnen „keys“ innerhalb einer „sections“ eindeutig sind.

Ein kleines Beispiel mag das verdeutlichen:

```
[Grundfarben]
rot=0xff0000
gruen=0x00ff00
blau=0x0000ff

[Lieblingsfarben]
schwarz=0x000000
rot=0xff0000
```

Um diese Eindeutigkeit und die Schreib- und Lesevorgänge kümmert Windows sich selbst, sodass Sie in AutoIt nur die Parameter angeben müssen.

5.6.1.1 Speichern von Werten in einer Ini-Datei

Zum Schreiben eines Wertes in eine Ini-Datei verwenden Sie die Funktion **`IniWrite`**. Diese hat folgende Syntax:

Listing 5.7: Syntax von IniWrite

```
IniWrite("filename", "section", "key", "value")
```

Um dem obigen Beispiel also die Lieblingsfarbe Blau hinzuzufügen, wäre folgender Aufruf nötig:

Listing 5.8: Beispiel zu IniWrite

```
1 IniWrite(@ScriptDir & "\farben.ini", "Lieblingsfarben", "
    blau", "0x0000ff")
```

Es ist *unbedingt* anzuraten, den Pfad komplett anzugeben. Es ist möglich, dass eine Angabe nur mit dem Dateinamen im Skriptverzeichnis ausgeführt wird. Wird das Skript aber in einem anderen Kontext gestartet, kann dies auch ein beliebiger anderer Pfad sein. Eine solche Fehlersuche ist sehr mühsam, daher der Rat, den Pfad immer mit anzugeben. Auch im Sinne einer sauberen Programmierung ist dies geboten.

Existiert die angegebene Datei nicht, so legt Windows sie automatisch an. Die Dateiendung „.ini“ ist nur notwendig, damit die Datei im Windows Explorer korrekt angezeigt wird. Liest und schreibt nur Ihre Anwendung die Dateien, können Sie auch eine andere Dateiendung vergeben.

Mit `IniWriteSection` lässt sich eine ganze section auf einmal aus einem Array oder einer Zeichenkette befüllen. Dabei ist die Schreibperformance geringfügig besser.

5.6.1.2 Lesen von Werten aus einer Ini-Datei

Fast analog zum Schreiben von Werten funktioniert auch das Auslesen der Werte aus der Datei. Die Funktion heißt konsequenterweise **`IniRead`**. Sie verlangt wie **`IniWrite`** als erste Parameter `filename`, `section` und `key`. Der vierte Parameter gibt an, welchen Wert die Funktion zurückgeben soll, wenn der Schlüssel nicht gefunden wird. Diesen kann man einerseits als Standardwert verwenden (beispielsweise bei Einstellungen), aber gleichzeitig auch für die Fehlererkennung:

Listing 5.9: Standardwerte bei IniRead

```
1 If IniRead(@ScriptDir & "\farben.ini", "Lieblingsfarben", "
    rot", "nein") =
2 "nein" Then MsgBox(0, "", "Rot ist keine Lieblinsfarbe")
```

Auch wenn man den genauen Aufbau einer Ini-Datei nicht kennt oder nicht vorhersagen kann, kommt man an die Daten heran: Zunächst kann man sich mittels **`IniReadSectionNames`** die Namen der einzelnen sections als Array zurückgeben lassen. Für jeden Abschnitt einzeln liefert die Funktion **`IniReadSection`** dann die Schlüssel-Wert-Paare in einem zweidimensionalen Array.

5.6.1.3 Werte aus Ini-Dateien löschen

Zum Löschen eines einzelnen Wertes aus einer Ini-Datei verwendet man den Befehl **IniDelete** mit drei Parametern:

Listing 5.10: IniDelete für einen Wert

```
1 IniDelete(@ScriptDir & "\farben.ini", "Lieblingsfarben", "
    schwarz")
```

Soll der ganze Abschnitt gelöscht werden, so lässt man den dritten Parameter einfach weg. Alle Lieblingsfarben tilgt man also folgendermaßen:

Listing 5.11: IniDelete für einen Abschnitt

```
1 IniDelete(@ScriptDir & "\farben.ini", "Lieblingsfarben")
```

5.6.2 Registry

Die Windows Registry wurde von Microsoft als Nachfolger der Ini-Dateien ersonnen. Sie soll an zentraler Stelle Einstellungen versammeln. Viele davon lassen sich nur hier umstellen, sodass für Ihr Skript möglicherweise nicht nur das Speichern und Lesen eigener Daten relevant ist, sondern auch der Zugriff auf fremde Einstellungen.

Ähnlich wie bei Ini-Dateien gibt es eine vorgegebene Struktur in der Daten eindeutig abgelegt werden. Hierarchisch geordnete Schlüssel und Unterschlüssel bilden die Basis. Jeder Schlüssel kann beliebig viele Einträge mit zugehörigen Werten haben. Das besondere ist, dass in der Registry verschiedene Datentypen für diese Einträge existieren. So gibt es REG_BINARY (für Binärdaten [?]), REG_SZ (für normale Zeichenketten), REG_MULTI_SZ (mehrzeilige Zeichenkette [?]), REG_EXPAND_SZ ([?] Zeichenkette) sowie REG_DWORD (Ganzzahlwert).

5.6.2.1 Werte schreiben

Die Syntax für das Schreiben von Daten in die Registry lautet wie folgt:

Listing 5.12: Syntax von RegWrite

```
1 RegWrite("keyname" [,"valuename", "type", value])
```

Die Basis-Schlüssel des keyname können in abgekürzter Form geschrieben werden (HKLM statt HKEY_LOCAL_MACHINE, HKU statt HKEY_USERS, HKCU statt HKEY_CURRENT_USER, HKCR statt HKEY_CLASSES_ROOT und HKCC statt HKEY_CURRENT_CONFIG.

Für type ist einer der oben genannten Typen einzusetzen.

Ein Beispiel:

Listing 5.13: Beispiel für RegWrite

```
1 RegWrite("HKEY_LOCAL_MACHINE\SOFTWARE\Test", "TestKey", "
    REG_SZ", "TestWert")
```

Diesen Wert können Sie entweder per Hand im Registry-Editor (Start -> Ausführen -> „regedit“) oder mit dem Befehl RegDelete wieder löschen (siehe [ref]).

5.6.2.2 Werte lesen

Einen bestimmten Wert aus der Registry zu lesen ist in AutoIt kein Problem. Der Befehl **RegRead** ist dafür zuständig und hat folgende Aufrufkonvention:

Listing 5.14: Syntax von RegRead

```
1 RegRead ("keyname", "valuename")
```

Die Hinweise für die Parameter gelten analog zu denen für **RegWrite**.

Das Lesen aller Unterschlüssel oder Einträge eines Schlüssels ist nicht ganz so trivial wie bei Ini-Dateien. Bei den Registry-Funktionen muss man dazu eine For-Schleife bemühen und selbst mit dem Rückgabewert feststellen, wann die Auflistung beendet ist. RegEnumKey liefert den Unterschlüssel mit einer bestimmten Nummer zurück, RegEnumVal die Einträge.

5.6.3 Dateien

Die größte Flexibilität bei der Verwaltung der eigenen Daten erhält man, wenn man sich selbst darum kümmert, sie in einer Datei zu hinterlegen und von dort wieder auszulesen.

Damit ist man an keine Konventionen hinsichtlich der Länge der Einträge oder des Formats gebunden, was im Umkehrschluss auch die Verantwortung mit sich bringt, sich selbst Gedanken darüber zu machen.

5.6.3.1 Daten speichern

Um einen Text in eine Datei schreiben zu können, müssen Sie sich zunächst ein sogenanntes *Handle* auf die Datei besorgen. Darüber gewährt das Betriebssystem den exklusiven Zugriff auf die Datei, das bedeutet, dass sie damit jederzeit eindeutig ansprechbar

ist. Dieses Handle liefert die Funktion **FileOpen**. Zu beachten ist dabei die richtige Wahl des Parameters mode. Er steuert, ob die Datei zum Lesen oder zum Schreiben geöffnet wird und ob der vorherige Inhalt gelöscht wird. Ist die Datei noch nicht vorhanden, so legt FileOpen sie beim Schreiben automatisch an.

Es ist *nicht* möglich, eine Datei gleichzeitig zum Lesen und Schreiben zu öffnen, da dabei nicht sichergestellt werden könnte, dass immer vorhersehbare Ergebnisse erzielt würden.

Mit dem Handle ist es nun möglich, Daten zeichen- oder zeilenweise in die Textdatei zu schreiben. Zu diesem Zweck gibt es die Funktionen **FileWrite** bzw. **FileWriteLine** . Als ersten Parameter übergibt man beiden das Handle auf die Datei.

Zum Abschluss des Dateizugriffes muss die Datei wieder freigegeben werden, damit andere Programme darauf zugreifen können. Das geschieht mit der Funktion **FileClose** .

Listing 5.15: FileOpen, FileWrite, FileWriteLine

```
1 $file_hwnd = FileOpen(@ScriptDir & "\datei.txt", 2) ; mode
     2: Alten Inhalt loeschen
2 FileWrite($file_hwnd, "Du bist ")
3 FileWriteLine($file_hmnd, "Deutschland.")
4 FileClose($file_hwnd)
```

[Bitwerte]

5.6.3.2 Daten lesen

Das Lesen aus einer Textdatei erfolgt vom Prinzip her analog zum Schreiben: Wieder benötigen Sie ein Handle der Datei, das dann an die folgenden Funktionen übergeben und am Ende wieder freigegeben werden muss.

Die Befehle zum Lesen heißen entsprechend **FileRead** und **FileReadLine**. Diese geben den ausgelesenen Text als Rückgabewert wieder, den Sie dann auffangen können:

Listing 5.16: FileOpen, FileRead, FileReadLine

```
1 $file_hwnd = FileOpen(@ScriptDir & "\datei.txt", 0) ; mode
     0: Lesen
2 $text = FileRead($file_hwnd, 7) ; enthaelt "Du bist"
3 $text_zeile = FileReadLine($file_hmnd, 1)
4 MsgBox(0, "Ergebnis", $text_zeile)
5 FileClose($file_hwnd)
```

Die Zählung der Zeilen beginnt im Gegensatz zu den meisten anderen Füllen bei eins und nicht bei null.

5.6.3.3 Kurzformen

Für Dateizugriffe, die nur aus einem oder sehr wenigen Lese- oder Schreibbefehlen bestehen, gibt es eine Kurzform, die ohne explizite Handles auskommt. Man gibt als ersten Parameter der eigentlichen Funktionen dazu einfach statt des Handles den Pfad zur Datei an. AutoIt öffnet und schließt die Datei dann selbstständig.

Ein Beispiel zur Verdeutlichung:

Listing 5.17: Dateizugriff - Kurzform

```
1 $text_zeile = FileRead(@ScriptDir & "\datei.txt")
2 MsgBox(0, "Ergebnis", $text_zeile)
3 FileClose($file_hwnd)
```

5.6.4 Vergleich: Ini-Dateien – Registry – Textdateien

Jede der drei vorgestellten Methoden für das Speichern von Daten außerhalb Ihres Skriptes hat einige Vorteile, die bei der Konzeption des Skriptes im Einzelfall bedacht werden sollten:

Ini-Datei	Registry	Textdatei
Eindeutigkeit	Eindeutigkeit	Flexibilität
Auflistung leicht	schneller Zugriff	eigene Datenstruktur möglich
einfach zu sichern	zentral gespeichert	einfach zu sichern
veraltet	modern	

Tabelle 5.12: Datenspeicherung im Vergleich

5.7 FileInstall

In vielen Fällen ist es wünschenswert, alle Dateien, die ein Programm braucht, in einer einzigen Datei zu verstauen, denn diese lässt sich wesentlich bequemer verteilen.

Für dieses Szenario bietet AutoIt die Funktion **`FileInstall`**. Deren Syntax lautet:

Listing 5.18: Syntax FileInstall

```
1 FileInstall("source", "dest" [, flag])
```

Die Besonderheit dieser Funktion ist, dass sie **zum ersten Mal** beim Kompilieren des Skriptes ausgeführt wird. Dabei wird die Datei, die als source angegeben ist, mit dem Skript zusammen zu einer Exe-Datei verarbeitet, die quasi die soure-Datei im Bauch mit sich führt. Weil der Compiler aber nicht wie AutoIt selbst in der Lage ist – und auch nicht versetzt werden soll – Variablen und Makros auszuwerten, ist es *unbedingt* erforderlich, den Pfad als einfache Zeichenkette anzugeben.

Die **zweite Bedeutung** erlangt **`FileInstall`** bei der Ausführung der entstandenen Exe-Datei: Dann wird die Datei, die sie mitführt, in das Verzeichnis, das als dest-Parameter angegeben wurde, kopiert. Da dieser Teil vom AutoIt-Interpreter selbst übernommen wird, können Sie hier ganz normal Variablen und Makros verwenden. Der letzte Parameter steuert schließlich, ob vorhandene Dateien überschrieben werden sollen.

5.7.1 Datum & Zeit

Funktionen, die mit Datum und Zeit arbeiten, sind fehlerträchtig. Das liegt vor allem daran, dass es eine Menge Sonderfälle gibt, die beachtet werden müssen. Dazu zählen nicht nur Schaltjahre, sondern auch die verschiedenen Monatslängen und jeder einzelne Tageswechsel.

Glücklicherweise bleibt es Ihnen in den allermeisten Fällen erspart, sich um alle diese Ausnahmen und Besonderheiten selbst zu kümmern, denn AutoIt bringt bereits eine Menge Funktionen für diese Fragestellungen mit.

Für deutsche (oder auch andere nicht-amerikanische) Nutzer eröffnet sich aber ein neues Problemfeld: Die Daten werden im amerikanischen Format verarbeitet, weswegen nach dem Einlesen vom Benutzer und vor dem Ausgeben oftmals Umwandlungen nötig werden. Lassen Sie sich aber davon nicht abschrecken, die mitgelieferten Funktionen sind trotzdem erste Wahl für die Arbeit.

Das amerikanische Datumsformat wird als mm/dd/yyyy dargestellt, will heißen zweistelliger Monat (z.B. 07 für Juli), zweistelliger Tag (z.B. 04) und vierstelliges Jahr (z.B. 1988). Getrennt werden die Angaben durch das Zeichen /.

Das Zeitformat lautet hh:mm:ss, also jeweils zweistellige Stunden, Minuten und Sekunden, abgetrennt mit Doppelpunkten. Es wird das 24-Stunden-Format verwendet, sodass die Zeit nicht für deutsche Gepflogenheiten umgewandelt werden muss.

Die Hilfe macht bei jeder Funktion Angaben darüber, in welchem Format die Zeiten angegeben werden müssen.

Die am häufigsten verwendeten Funktionen aus dieser Gruppe sind _DateAdd und _DateDiff. Erstere „addiert“ zu einem Datum einen Zeitraum. Man kann also auf einfache Weise ausrechnen, wie spät es „in 15 Minuten“ oder „in 5 Tagen“ ist.

[Beispiel DateAdd]

Die Funktion _DateDiff vergleicht zwei Daten und gibt den Abstand (je nach Wunsch in Minuten, Stunden, Tagen usw.) zurück. So können Sie auch überprüfen, welches Datum älter ist.

[Beispiel DateDiff]

Wie in den vorangegangenen Beispielen bereits verwendet, kann man das aktuelle Datum und die aktuelle Zeit auch über Funktionen ermitteln. Dazu stehen „_NowCalc“ (Datum und Zeit im amerikanischen Format), „_NowCalcDate“ (Datum amerikanisch) und „_NowTime“ (Zeit) zur Verfügung.

Die Umwandlung in das Format, dass der Computernutzer bei sich eingestellt hat, nimmt die Funktion _DateTimeFormat vor. Im zweiten Parameter legen Sie das Ausgabeformat fest. 1 steht für ein langes Datumsformat, 2 ein kurzes; 3 macht die Zeit kurz (ohne Sekunden), 4 lang.

[todo]

5.8 Eigene Funktionen

Für viele, aber nicht alle Fälle bietet AutoIt vorgefertigte Funktionen an. Suchen Sie eine Funktion vergeblich, so können Sie sich selbst Funktionen schreiben, die sich genau so aufrufen und steuern lassen, wie die internen.

5.8.1 Grundbau und Aufruf

Eine eigene Funktionsdefinition sieht in AutoIt folgendermaßen aus:

Listing 5.19: Grundbau einer eigenen Funktion

```
1 Func meineFunktion()
2     MsgBox(0, "", "meineFunktion() aufgerufen")
3 EndFunc
4
5 meineFunktion()
```

Die Schlüsselworte **Func** und **EndFunc** schließen die Funktion ein. Hinter **Func** steht der Name, unter dem die Funktion aufgerufen werden kann.

Aufgerufen wird die neu geschriebene Funktion über eben diesen Namen, gefolgt von runden Klammern.

Die Benennung der Funktion ist reglementiert: Es dürfen nur Groß- und Kleinbuchstaben, Zahlen und der Unterstrich verwendet werden. An der ersten Stelle des Namens sind Zahlen nicht erlaubt.

Es ist nicht möglich, eigene Funktionsdefinitionen zu verschachteln, zwischen zwei **Func** s *muss* also immer ein **EndFunc** stehen.

Auch innerhalb von **If**-, **Select**- und **Switch**-Blöcken sowie allen Arten von Schleifen können keine Funktionen definiert werden. Ansonst ist die Platzierung nicht relevant, Funktionen können auch vor ihrer Definition aufgerufen werden.

5.8.2 Parameter

Ebenso wie in der Mathematik zeichnet sich eine Funktion dadurch aus, dass sie einen (oder mehrere) Werte aufnehmen kann und Aktionen in Abhängigkeit von diesen ausführt. Diese Werte nennen sich Parameter und lassen sich auch für eigene Funktionen definieren.

Ein einfacher Parameter wird einfach in den Klammern hinter der Funktionsdefinition angegeben. Unter dem Namen, der hier gewählt wird, ist der übergebene Wert innerhalb der Funktion ansprechbar – unabhängig davon, welche Zeichenkette oder Variable beim Aufruf verwendet wurde.

Listing 5.20: Einfache Parameter

```
1 Func Log($text)
2     MsgBox(0, "", $text)
3 EndFunc
4
5 Log("Error 404")
```

Wie bei AutoIt-internen Funktionen kann man auch bei eigenen Funktionen Parameter

als optional markieren – dann muss aber auch ein Wert angegeben werden, der verwendet wird, wenn der Parameter beim Aufruf ausgelassen wird.

Wichtig ist, dass man Parameter nur von hinten her optional machen kann. Das heißt der vorletzte Parameter kann nur optional sein, wenn es auch der letzte ist. Das liegt daran, dass AutoIt nur so sicher sein kann, welcher Parameter weggelassen wurde.

Ein Beispiel mag das verdeutlichen:

Listing 5.21: Optionale Parameter

```
1 Func Fehler($title, $text = "Allgemeiner Fehler")
2     MsgBox(0, $title, $text)
3 EndFunc
4
5 Fehler("Fehler", "Fehler 503")
6 Fehler("Error")
```

Ein dritter Aspekt sind Parameter, die als Referenz übergeben werden, das bedeutet, dass die übergebenen Variablen direkt bearbeitet werden. Am Ende der Funktion wird der Wert der mit ByRef übergebenen Variablen zurückgeschrieben.

Listing 5.22: Parameter mit ByRef

```
1 Func Summe(ByRef $a, $b)
2     $a = $a + $b
3 EndFunc
4
5 Global $zahl1 = 11
6 Global $zahl2 = 31
7 Summe($zahl1, $zahl2)
8 MsgBox(0, "Ergebnis", $zahl1)
```

In dem Makro @NumParams wird übergeben, wie viele Parameter die Funktion beim Aufruf tatsächlich erhalten hat. Damit ist in Einzelfällen eine noch genauere Unterscheidung der verschiedene Aufrufe möglich.

5.8.3 Rückgabewert, `@error` und `@extended`

Mathematische Funktionen haben eine weitere Gemeinsamkeit mit den Funktionen, die Sie in AutoIt schreiben können: Es gibt die Möglichkeit, das Ergebnis einer Berechnung oder eine Meldung über Erfolg oder Misserfolg zurückzugeben.

Zum einen gibt es dazu den Rückgabewert, den man über das Schlüsselwort **Return** einleitet. An dieser Stelle wird die Funktion beendet und der angegebene Ausdruck

zurückgegeben. Der Funktionsaufruf wird dabei durch den Rückgabewert ersetzt.

Man kann sich also vorstellen, dass AutoIt zunächst die Funktion auswertet und durch ihren Rückgabewert ersetzt. Danach wird der Ausdruck weiter ausgewertet. Durch dieses Vorgehen kann man sehr kompakten Code schreiben:

Listing 5.23: Rückgabewert einsetzen

```
MsgBox(0, "", "Summe: " & Summe(10, 32))

Func Summe($a, $b)
    Return $a + $b
EndFunc
```

Treten Fehler auf, möchte man diese nicht immer umständlich über den Rückgabewert codieren müssen. Besser wäre ein eigener „Kanal“, über den man Fehler melden könnte. Dazu gibt es das Makro `@error`. Um dessen Wert zu setzen, verwenden Sie die Funktion SetError:

Listing 5.24: @error verwenden

```
$ergebnis = Division(81, 9)
If @error = 1 Then
   MsgBox(0, "", "Fehler: Division durch Null.")
Else
   MsgBox(0, "", "Quotient: " & $ergebnis)
EndIf

$ergebnis = Division(81, 0)
If @error = 1 Then
   MsgBox(0, "", "Fehler: Division durch Null.")
Else
   MsgBox(0, "", "Quotient: " & $ergebnis)
EndIf

Func Division($a, $b)
          If $b = 0 Then
             SetError(1)
             Return ""
          Else
             Return $a / $b
          EndIf
EndFunc
```

Wichtig ist, dass direkt nach SetError Return aufgerufen wird, da sonst der @error-Wert von einer anderen Funktion überschrieben werden kann.

Analog kann man Informationen, die über den Rückgabewert hinausgehen, im Makro @extended zurückliefern. Hierfür eignet sich die Funktion SetExtended. Sollen sowohl @error als auch @extended mit Werten versehen werden, so rufen Sie SetError mit zwei Parametern auf.

5.9 Fehlerbehandlung

Ein beträchtlicher Teil der Arbeit beim Programmieren besteht darin, sich damit auseinanderzusetzen, welche Fehler auftreten *könnten* und wie darauf zu reagieren ist. Die schlechteste Reaktion ist Verschweigen – es sollte für den Anwender zumindest erkenntlich sein, *dass* etwas schief gelaufen ist. Besser noch wäre es, ihm konkret den Fehler und mögliche Ursachen und deren Behebung zu nennen.

Im Wesentlichen muss man dazu bei AutoIt auf die im [ref return] Rückgabewert und in @error hinterlegten Informationen zurückgreifen. Anhand dieser Werte sind die Probleme abzulesen, die eine Funktion während ihrer Durchführung hatte.

Die Funktion FileOpen gibt -1 zurück, wenn die Datei nicht geöffnet werden kann. Eine Abfrage sähe also so aus:

Listing 5.25: Fehlerbehandlung mit Rückgabewert

```
1 $file = FileOpen(@ScriptDir & "\test.txt", 0)
2
3 If $file = -1 Then
4    MsgBox(0, "", "Datei kann nicht geoeffnet werden.")
5 Else
6          MsgBox(0, "Dateiinhalt", FileRead($file))
7    FileClose($file)
8 EndIf
```

Andere Funktionen geben Ihren Status über das Makro `@error` zurück, so beispielsweise **`FileSelectFolder`**:

Listing 5.26: Fehlerbehandlung mit `@error`

```
1 $dir = FileSelectFolder("Bitte auswaehlen", "C:\Windows")
2
3 If @error Then
4         MsgBox(0, "Fehler", "Abgebrochen.")
5 Else
6         MsgBox(0, "Ordner", $dir)
7 EndIf
```

AutoIt-Quelltext kann leider schnell unübersichtlich werden, wenn viele dieser Abfragen ineinander verschachtelt werden müssen. Hier ist das Konzept objektorientierter Sprachen wie C++ oder C# deutlich leistungsfähiger. Dennoch lassen sich alle Fehler korrekt verarbeiten und Ihr Skript kann wie gewünscht auf alle entstehenden Situationen reagieren, wenn es darauf vorbereitet ist.

5.10 Stringverarbeitung

Es gibt kaum ein größeres Skript, in dem nicht irgendwann die Frage auftaucht, wie man Zeichenketten den eigenen Wünschen gemäß manipulieren kann. Für diese Fragestellung bietet AutoIt eine ganze Menge Funktionen, die Ihnen fast alle Freiheiten lassen.

Zunächst lassen sich *Eigenschaften* einer Zeichenkette herausfinden: Die Funktion StringLen liefert die Anzahl der enthaltenen Zeichen, eine Reihe weiterer Funktionen prüft, ob ein String bestimmte Formate erfüllt, nämlich **StringIsAlNum**, **StringIsAlpha**, **StringIsASCII**, **StringIsDigit**, **StringIsFloat**, **StringIsInt**, **StringIsLower**, **StringIsSpace**, **StringIsUpper** sowie **StringIsXDigit**.

Listing 5.27: Stringfunktionen – Eigenschaften

```
$test = "A23 nach Bonn"
For $i = 1 to StringLen($test)
    $zeichen = StringMid($test, $i, 1)
    MsgBox(0, $zeichen, "Zahl? " & StringIsDigit($zeichen) _
       & "Grossbuchstabe? " &
@CRLF & StringIsUpper($zeichen))
Next
```

Mit dem Befehl StringInStr können Sie in einer Zeichenkette nach einer anderen suchen. Rückgabewert ist die Position des gesuchten Teiles.

Mit StringMid lassen sich Zeichenketten „zerschneiden“. Als Parameter geben Sie neben dem zu bearbeitenden String die Startposition und die Anzahl der Zeichen an, die zurückgegeben werden sollen.

Folgendes Beispiel sucht alle *Es* in einem Text und zeigt es mit dem Zeichen davor und danach an.

Listing 5.28: Stringfunktionen – Funktionen

```
$test = "Wissenschaftler haben festgestellt..."
Dim $i = 1
While 1
    $pos = StringInStr($test, "e", 0, $i)
```

```
        If $pos = 0 Then ExitLoop
    $teil = StringMid($test, $pos-1, 3)
    MsgBox(0, $i, $teil)
    $i += 1
WEnd
```

Eine weitere häufig verwendete Funktion ist StringSplit. Damit lässt sich eine Zeichenkette an einem oder mehreren Zeichen in Einzelteile zerlegt, die in einem Array abgelegt werden. Dort lassen sich viele Daten dann besser weiterverarbeiten:

Listing 5.29: StringSplit

```
#include <Array.au3>
$test = "Stefan Mueller,Kochweg 25,45842 Gelsenkirchen"
$teile = StringSplit($test, ",")
_ArrayDisplay($teile, "Adresse aufgeteilt")
```

Dazu kommen reichlich Funktionen, mit denen man Zeichenketten verändern kann. Sie sind in Kurzform in der folgenden Tabelle aufgelistet:

Funktion	Beschreibung	Mögliches Ergebnis
Abschneiden		
StringTrimLeft	Von links abschneiden	[est text]
StringTrimRight	Von rechts abschneiden	[Test tex]
StringStripWS	Leerzeichen entfernen	[Test text]
StringStripCR	Zeilenumbrüche entfernen	—
Ersetzen		
StringReplace	Ersetzt einen Text durch einen anderen	[Text text]
Schreibweise		
StringUpper	In Großbuchstaben verwandeln	[TEST TEXT]
StringLower	In Kleinbuchstaben verwandeln	[test text]

Tabelle 5.14: Zeichenkettenfunktionen

Weitere Funktionen stellt die UDF „String.au3“ bereit. Highlights dieser Sammlung ist die Ver- und Entschlüsselung in Art der RC4-Methode mittels _StringEncrypt sowie die Funktion _StringBetween, die beim Parsen**[Parsen??]** von Texten oft gute Dienste leistet.

Listing 5.30: _StringBetween

```
#include <String.au3>
$test = "<head><title>Neue Seite 1</title></head>"
$title = _StringBetween($test, "<title>", "</title>")
_ArrayDisplay($title, "Titel")

MsgBox(0, "", _StringEncrypt(1, "Test", "passwort")
```

Noch weiter gehende Funktionen zur Arbeit mit Zeichenketten eröffnen die Regulären Ausdrücke (engl. Regular Expressions, kurz RegEx), die später besprochen werden [ref regex].

5.10.1 StringFormat

5.11 Explizite Typenumwandlungen

Normalerweise müssen Sie sich in Ihren Skripten nicht darum kümmern, ob Variablen als Zahlen oder Zeichenketten behandelt werden. AutoIt erkennt am Kontext, was benötigt wird und nimmt die Konvertierung selbst vor. Intern liegt die Variable in einem sogenannten Variant-Datentyp vor, aus dem dann der entsprechende Datentyp zur Laufzeit generiert wird.

Es ist aber auch möglich, diese Konvertierung zu erzwingen. Dazu gibt es für jeden Typ eine Funktion, im einzelnen sind das String für Zeichenketten, Number für Zahlen allgemein, Int für Ganzzahlen, und BinaryString für Binärdaten.

Der Vorteil der Konvertierung „von Hand“ ist, dass danach *sicher* der gewünschte Typ vorhanden ist – es können also eine ganze Reihe von Fehlern verhindert werden, die durch falsche Eingaben oder Werte in Variablen ausgelöst werden.

5.12 Tidy

Falls Sie wie empfohlen den Editor SciTE4AutoIt installiert haben, können Sie Quelltext mit der Tastenkombination Strg+T bequem, schnell und vollautomatisch einrücken lassen. Das verändert zwar nicht die Skriptfunktion, macht es aber wesentlich leichter zu lesen, denn die Strukturen werden auf den ersten Blick klar.

Die Einrückungspraxis ist wie erwähnt zwar nicht zwingend erforderlich, aber auch in anderen Programmiersprachen wird nach den selben Regeln eingerückt, die Tidy anwendet.

Es ist daher ratsam, jeden Quelltext, den man weitergeben möchte, zuvor mit Tidy in ein Format zu bringen, dass es anderen nicht unnötig erschwert, ihn zu verstehen.

5.13 Fehlersuche

[3]

5.14 Fehlermeldungen

Am Anfang passieren jedem Anwender einer für ihn neuen Programmiersprache viele Fehler – Fehler, die, wenn man weiß, woran es liegt, leicht zu beheben sind. So ist es natürlich auch bei AutoIt.

5.14.1 zur Kompilierzeit

Ein leistungsfähiges Tool namens AU3Check untersucht Ihren Quelltext, bevor Sie ihn aus SciTE heraus starten. Die Fehlermeldungen, die AutoIt ausgibt, sind im Prinzip sehr präzise und hilfreich zur Behebung des Fehlers, aber dennoch verwirren sie anfangs oft. Daher hier eine Auswahl häufiger Fehler und der dazu gehörigen Meldungen:

5.14.1.1 ERROR: MsgBox() [built-in] called with wrong number of args.

AutoIt beschwert sich in dieser Form, wenn Sie eine mitgelieferte Funktion mit zu wenigen oder zu vielen Parametern aufrufen:

Listing 5.31: Fehlermeldung „wrong number of args“

```
1 MsgBox(0, "Titel")
2 MsgBox(0, "Titel", "Text", 20, "fuenf")
```

Wie Sie inzwischen wissen erfordert eine MsgBox drei oder vier Parameter, aber eben nicht zwei oder fünf.

Beheben lässt sich dieser Fehler, indem man eine korrekte Anzahl von Parametern übergibt (siehe [ref]).

5.14.2 zur Laufzeit

Es gibt aber auch Fehler, die erst zur Laufzeit des Programmes auftreten und dann zu einer entsprechenden Meldung und in der Regel zum Absturz des Programmes führen. Diese sind schwerer zu entdecken, da sie nicht unter allen Umständen auftreten, sondern oft nur unter ganz bestimmten Voraussetzungen und bei bestimmten Aktionen:

5.14.3 Array variable has incorrect number of subscripts or subscript dimension range exceeded.

Dieser Fehler tritt recht häufig auf. Er wird dadurch ausgelöst, dass ein Array mit einem Index aufgerufen wird, der nicht existiert. Ein einfaches, aber lehrreiches Beispiel:

Listing 5.32: Fehlermeldung „dimension range exceeded“

```
1 Dim $array[10]

3 For $i = 1 to 10
4         ToolTip($array[$i])
5 Next
```

Das Problem ist hier, dass im letzten Durchlauf der For-Schleife der Wert von `$array[10]` ausgelesen werden soll. Das kann aber nicht funktionieren, da das Array mit zehn Elementen nur die Elemente 0 bis 9 kennt.

Die Nullbasiertheit der Arrays ist auch der Grund dafür, dass man eine For-Schleife, die ein Array komplett durchlaufen soll, von 0 bis `UBound($array)-1` ansetzt.

Dieser Fehler ist weit tückischer, wenn man Array-Größen häufig verändert oder gänzlich dynamisch verwaltet.

5.15 Besonderheiten unter Windows Vista

[0.5] @Programfilesdir #requireadmin

6 Professionelle Anwendung

6.1 GUI

6.1.1 Generelle Hinweise

[2]

6.1.2 Koda

[1]

6.1.3 GUIs mit mehreren Fenstern

[1]

6.2 Threads

6.2.1 Kommunikation zwischen Skripten

6.3 Datenbanken mit SQLite

[3]

6.4 Reguläre Ausdrücke

[2]

6.5 IE.au3 - Fernsteuerung des Internet Explorers

[5]

6.6 FTP.au3

externe UDF! [4]

6.7 Netzwerkprogrammierung

[5]

7 Administrative Aufgaben

7.1 Arbeit mit DOS-Befehlen

Stdout [2]

7.2 Installationen automatisieren

[5] BlockInput!

7.3 Scriptomatic

[3]

8 Über den Tellerrand

8.1 AutoItX

[3]

8.2 PlugIn-Entwicklung

[5]

9 Beispielprojekt

[Startrampe?] - Passwort generieren basiernd auf URL

10 Frequently Asked Questions

Es gibt eine Reihe Fragen zu AutoIt, die immer wieder aufkommen. Einige davon sollen im Folgenden knapp beantwortet werden. Es wird dabei auf die Auflistung in der Hilfe-Datei zurückgegriffen, die mit zusätzlichen Fragen aus der Praxis ergänzt wurde.

10.1 Meine Skripte werden als Viren erkannt. Woran liegt das?

Von Zeit zu Zeit werden alle kompilierten AutoIt-Skripte von Virenscannern als vermeintliche Trojaner oder anderes Ungeziefer erkannt.

Der Grund dafür ist, dass es Menschen gibt, die AutoIt verwenden, um damit Schadsoftware zu programmieren. Wenn nun die Antivirenexperten auf ein solches Skript stoßen, dann erstellen sie eine Signatur, quasi einen Fingerabdrück der Datei. Für den Fall, dass diese Signatur aus dem Bereich genommen wird, der den AutoIt-Interpreter enthält, bedeutet das, dass alle kompilierten Skripte diesem Schädlingstyp zugeordnet und moniert werden.

Abhilfe schafft in der Regel eine Beschwerdemail an Ihren Antivirensoftwarehersteller. Für die Zeit bis zu einer entsprechenden Reaktion bleibt nur, das Verzeichnis mit Ihren AutoIt-Skripten von der Prüfung auszunehmen sofern das möglich ist.

10.2 Wie kann ich eigene Texte im Eigenschaften-Dialog der Exe-Datei unterbringen?

Um die Daten, die angezeigt werden, wenn man die Eigenschaften der kompilierten Datei im Explorer anzeigen lässt, anpassen zu können, benötigen Sie das Programm „Resource Hacker“ von http://www.angusj.com/resourcehacker/. Nach dessen Installation stehen die zusätzlichen Informationen zur Verfügung, wenn Sie das Skript mit der rechten Maustaste im Explorer anklicken und „Compile with options“ wählen.

10.3 Wie kann ich Kommandozeilenparameter verarbeiten?

Beim Skriptstart wird jedesmal ein Array namens `$cmdline` angelegt. Das erste Element (mit dem Index 0) enthält die Anzahl der übergebenen Parameter, die folgenden Elemente enthalten dann jeweils einen dieser Parameter.

Zudem wird eine Variable namens `$CmdLineRaw` mit dem kompletten Programmaufruf mit Parametern gefüllt. Diese Variable können Sie dann selbst verarbeiten.

Listing 10.1: Kommandozeilenparameter verarbeiten

```
1 If $CmdLine[0] > 0 Then ; ueberhaupt Parameter uebergeben?
2   If $CmdLine[1] = "-help" Then MsgBox(0, "", "Wofuer?")
3 EndIf
```

Der Aufruf des kompilierten Skriptes müsste dann wie folgt aussehen:

Listing 10.2: Kommandozeilenparameter übergeben

```
1 ShellExecute("kommando.exe", "-help")
```

10.4 Wie kann ich die Größe der Exe-Datei minimieren?

Es gibt zwei Programme, die anbieten, Ihr Skript von unbenutzten Funktionen (viele davon werden aus den UDFs geladen) und Variablen zu säubern. Dazu können Leerzeilen und Kommentare entfernt werden. Das reduziert die Größe des Quelltextes oft drastisch, und wegen der Interpreterstruktur von AutoIt [ref] auch die Größe der kompilierten Exe-Datei um einige Kilobytes.

- Au3Preprocessor von „kjactive“:
  http://www.autoitscript.com/forum/index.php?showtopic=38669
- Cleanscript von „Markus“ und „peethebee“:
  http://www.autoitscript.com/forum/index.php?showtopic=30573

10.5 Funktioniert AutoIt auch unter Linux?

Bedingt. Das liegt daran, dass AutoIt speziell für Windows entwickelt wurde und deshalb auch so angenehm dort funktioniert. Es gibt unter Linux das Programm „wine“, dass Windows-APIs emuliert und damit teilweise AutoIt-Skripte zum Laufen bringen kann.

Es gibt aber auch Alternativen wie „bash“, „Perl“, „Python“, „PHP“, die unterschiedliche Akzentuierungen haben.

10.6 Wie kann ich meine AutoIt Version herausfinden?

Listing 10.3: AutoIt-Version ausgeben

```
1 MsgBox(0, "", @AutoItVersion)
```

10.7 Wie kann ich Autoit automatisch updaten lassen?

Rufen Sie dazu die Datei C:\Programme\Autoit\Extras\AutoUpdateIt\AutoUpdateIt.au3 auf. Diese ermittelt selbstständig, ob Ihre AutoIt-Version aktuell ist und bietet andernfalls den Download der entsprechenden Version an.

10.8 Ein Fenster bei einer Installation erscheint nur manchmal. Wie fange ich es ab?

Es wird davon ausgegangen, dass Fenster 3 immer erscheint, Fenster 2 jedoch quasi optional ist, also erscheinen kann oder auch nicht.

Listing 10.4: Fenster abfangen

```
1  ; Fenster 1 verarbeiten
2  While 1
3      If WinExists("Titel Fenster 2", "Text Fenster 2") Then
4          ; auf Fenster 2 reagieren (ControlClick, Send, ...)
5      EndIf
6
7      If WinExists("Titel Fenster 3", "Text Fenster 3") Then
8          ; auf Fenster 3 reagieren (ControlClick, Send, ...)
9          ExitLoop
10     EndIf
11
12     Sleep(1000)
13 WEnd
14 ; Fenster 4 verarbeiten
```

10.9 Wie kann ich den Taskmanager aufrufen, obwohl ich Alt+Strg+Entf nicht simulieren kann?

Dazu gibt es zwei Möglichkeiten:

Die erste führt über die wenig bekannte, aber gleichwertige Tastenkombination Strg+Shift+ESC:

Listing 10.5: Strg+Alt+Entf ersetzen, Variante 1

```
1 Send("^+{ESC}")
```

Die andere Möglichkeit ist der direkte Aufruf des Taskmanagers mit ShellExecute:

Listing 10.6: Strg+Alt+Entf ersetzen, Variante 2

```
1 ShellExecute("taskmgr.exe")
```

10.10 Wo ist der „goto“-Befehl hingekommen?

Der von BASIC und AutoIt vor Version 3 bekannte Befehl `goto`, der innerhalb des Quelltextes an definierte Stellen springen konnte, ist in AutoIt 3 nicht mehr enthalten.

Auch wenn dies auf den ersten Blick als Verschlechterung erscheint, zwingt es zu strukturierterer Programmierung.

Denn statt dessen kann und sollte man Schleifen und Kontrollstrukturen einsetzen [ref]. Damit wird extremer „Spaghetti-Code“ vermieden und der Programmablauf leichter nachvollziehbar gestaltet.

10.11 Warum erhalte ich Fehlermeldungen, wenn ich doppelte Anführungszeichen verwende?

Wenn Sie in einer Zeichenkette das Zeichen `"` verwenden möchten, dann können Sie die Zeichenkette in einfache Anführungszeichen (`' '`) einschließen. Grundsätzlich sind einfache und doppelte Anführungszeichen gleichwertig.

Die andere Möglichkeit ist, das Zeichen `"` mit `""` zu escapen.

10.12 Warum kann ich Variablen nicht verwenden, indem ich folgendes schreibe: „Meine Variable hat den Wert $variable“?

Um dieses Verhalten zu aktivieren, müssen Sie die Option `Opt("ExpandVarStrings", 1)` setzen. Danach ist die Ausgabe der Variablen direkt in der Zeichenkette möglich, wenn sie von $ auf *beiden* Seiten begrenzt ist. Andere Variablen werden wie gewohnt angesprochen.

Listing 10.7: Anführungszeichen

```
Opt("ExpandVarStrings", 1)
Global $var = "Inhalt"
MsgBox(0, "", "Der $var$ zaehlt")
```

10.13 Warum passieren merkwürdige Dinge, wenn ich Variablen an den `Send`-Befehl übergebe?

Wenn Sie den Inhalt einer Variablen per Send-Befehl weitergeben, werden spezielle Zeichen wie !, `, + oder SPACE umgewandelt. Das ist aber nur selten erwünscht. Um das zu umgehen, benutzt man den „RAW“-Modus des Send-Befehls, der diese Umwandlung nicht durchführt, sondern die Zeichen so wie sie sind, simuliert:

Listing 10.8: Send im RAW-Modus

```
Local $meinevariable = "Cool! es geht!"
Send($meinevariable, 1)
```

10.14 Wie kann ich eine Datei mit einbinden?

siehe [ref FileInstall]

10.15 Was ist der Unterschied zwischen dem „return“-Wert und @error?

Normalerweise wird der „return“-Wert dazu benutzt, anzugeben, ob eine Funktion erfolgreich durchgeführt wurde. Gibt die Funktion (wie z.B. WinGetText()) aber schon

einen anderen Wert zurück, dann braucht man eine andere Möglichkeit, um festzustellen, ob die Funktion erfolgreich ausgeführt wurde. Dazu dient dann @error. [ref return, fehlerbehandlung]

10.16 Wie kann ich zwei GUIs parallel kontrollieren?

[ref]

10.17 Wie kann ich ein Skript nach einem Neustart fortsetzen?

Dazu können Sie sich eines Schlüssels in der Windows Registry bedienen, der ein Programm (in diesem Fall Ihr Skript), einmalig nach einem Neustart ausführt. Dieser Schlüssel findet sich unter `HKLM\Software\Microsoft\Windows\CurrentVersion\RunOnce` . Um dann an einer bestimmten Stelle fortfahren zu können, ist es möglich, einen Kommandozeilenparameter zu übergeben, den das Skript dann auswerten kann (siehe [ref]).

Listing 10.9: Skript nach Neustart fortsetzen

```
1  If ($CmdLine[0] < 1) Or ($CmdLine[1] <> "restarted") Then
2      ; normaler Ablauf vor dem Neustart
3      MsgBox(0, "Meldung", "Neustart wird eingeleitet...")
4  RegWrite("HKLM\Software\Microsoft\Windows\CurrentVersion\
       RunOnce",
5  "Beschreibung", "REG_SZ", '"' '\&' @ScriptFullPath '\&' '"
       restarted')
6      Shutdown(6) ; Erzwungener Neustart
7      Exit
8  Else
9      MsgBox(0, "Meldung", "Neustart abgeschlossen.")
10 EndIf
```

Dieses Vorgehen funktioniert aufgrund der Verwendung von `@ScriptFullPath` nur, wenn das Skript kompiliert – also als exe-Datei – ausgeführt wird.

10.18 Wie kann sich mein Skript selbst löschen?

Rufen Sie dazu einfach die Funktion _SelfDelete und anschließend **Exit** auf.

Hintergrund: Es wird eine .bat-Datei angelegt, die am Ende gestartet wird. Diese wartet die als ersten Parameter angegebene Zahl von Sekunden ab, bevor Sie zuerst das Skript und dann sich selbst löscht. Der Grund für diesen Umweg ist, dass sich unter Windows nur .bat-Dateien selbst löschen können, während sie ausgeführt werden.

Listing 10.10: Skript selbst löschen

```
Func _SelfDelete($iDelay = 0)
        Local $sCmdFile
        FileDelete(@TempDir & "\scratch.bat")
        $sCmdFile = 'ping -n ' & $iDelay & '127.0.0.1 > nul' & @CRLF & ':loop' &
@CRLF & 'del "' & @ScriptFullPath & '"' & @CRLF & 'if exist "' & @ScriptFullPath
& '" goto loop' & @CRLF & 'del ' & @TempDir & '\scratch.bat'
        FileWrite(@TempDir & "\scratch.bat", $sCmdFile)
        Run(@TempDir & "\scratch.bat", @TempDir, @SW_HIDE)
EndFunc
```

10.19 Wie kann ich mein Script über einen Hotkey beenden?

Listing 10.11: Skript über Hotkey beenden

```
HotKeySet("^!x", "MeinExit")

Global $i = 0
While $1
        $i = $i + 1
        ToolTip($i)
WEnd

Func MeinExit()
        Exit
EndFunc
```

10.20 Wie kann ich ein selbst gewähltes Icon benutzen, wenn ich mein Skript kompiliere?

Beim Kompilieren über das Kontextmenü im Explorer („Compile with options“) können Sie ein eigenes Icon für Ihr Programm auswählen. Dieses Logo muss dann nicht mit weitergegeben werden, es wird in die Exe-Datei mit integriert.

10.21 Wie kann ich sicher gehen, dass nur eine Instanz meines Skriptes läuft?

Die gewünschte Funktionalität implementiert die Funktion `_Singleton` aus der UDF „Misc.au3“. Sie erwartet als Parameter eine beliebige Zeichenkette, die das Skript eindeutig identifiziert:

Listing 10.12: Nur eine Instanz zulassen

```
#include <Misc.au3>
_Singleton("MeinSkript")
MsgBox(0, "", "Gestartet.")
```

10.22 Wo liegen die derzeitigen technischen Grenzen von AutoIt v3?

Sprachelement	Grenze
Einzelnen Skriptzeile	4.095 Zeichen
Zeichenkette	2.147.483.647 Zeichen
Fließkommazahl	1.7E±308 mit 15 Stellen Genauigkeit
Ganzzahl	64-bit vorzeichenbehaftet
Hexadezimalzahl	32-bit vorzeichenbehaftet
Arrays	64 Dimensionen und/oder 16 Mio. Elemente
Tiefe rekursiver Funktionsaufrufe	384 Ebenen
Gleichzeitig geöffnete Dateien	64
Gleichzeitig aktive Hotkeys	64
Gleichzeitig benutzte Variablen	unbegrenzt
Benutzerdefinierte Funktionen	unbegrenzt
GUI-Fenster	1024
Controls	4096 pro Fenster

Tabelle 10.2: Technische Grenzen von AutoIt

Oftmals sind diese Grenzen eher theoretischer Natur, das heißt, es können Performanceprobleme auftreten, bevor das Limit erreicht wird.

11 Danksagungen

Danke an alle, die zu diesem Buch beigetragen haben, insbesondere Rasta und Bernd, die Korrektur gelesen und wertvolle Verbesserungsvorschläge gemacht haben.

Auch **Jon** und den anderen Entwicklern von AutoIt und den zugehörigen Tools sei an dieser Stelle nochmals Dank ausgesprochen.

Dieses Buch ist mit **LaTeX** gesetzt worden. Für diese großartige Software danke ich allen, die daran beteiligt sind oder waren, insbesondere dem Erfinder von TeX **Donald E. Knuth**.

Daran möchte ich gleich den Dank an alle Entwickler von Freier Software anschließen, insbesondere für **Linux**.

Danke.

Tabellenverzeichnis

Listings