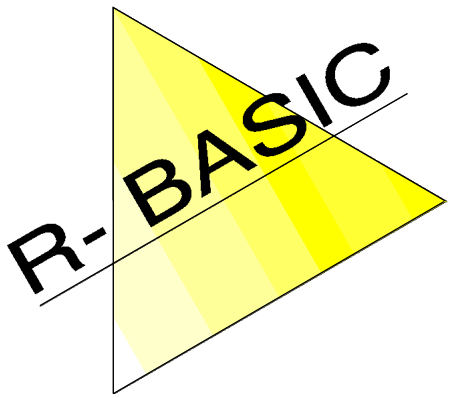
R-BASIC

Einfach unter PC/GEOS programmieren



Spezielle Themen

Volume 2
Dateien, Laufwerke, Ports,
Focus, Target, Edit-Menü

Version 1.0

(Leerseite)

Inhaltsverzeichnis

(Leerseite)

8 Verwaltung von Dateien

8.1 Kopieren, Verschieben und Löschen

Die R-BASIC Befehle zum Kopieren, Verschieben und Löschen von Dateien sind in ihrer Grundsyntax sehr einfach anzuwenden und tun ihren Job "so gut wie möglich" indem sie z.B. auch versteckte, System- oder schreibgeschützte Dateien löschen bzw. überschreiben. Durch die Möglichkeit Flagzeichen anzugeben kann der ambitionierte Programmierer ihr Verhalten aber sehr gut steuern und sich dadurch viel Programmierarbeit sparen. Am Ende des Abschnitts finden Sie eine Beschreibung des Konzepts, das hinter den Flags steht und eine Auflistung aller Flagzeichen und ihrer Bedeutung. Mögliche Fehlercodes für die Variable **fileError** finden Sie im Anhang.

FileCopy

FileCopy kopiert eine Datei. Das kann eine DOS oder eine GEOS-Datei sein.

Syntax: **FileCopy quelle$, ziel$** [, **flags$**]

quelle$: Bezeichnet die zu kopierende Datei. Pfadangaben sind zulässig
ziel$: Bezeichnet Ort und Namen, wohin die Datei kopiert werden soll. Pfadangaben sind zulässig. Es muss der Name der neuen Datei enthalten sein. Er darf vom Namen des Originals abweichen (automatisches Umbenennen beim Kopieren).
flags$: Optional: Zeichenkette. Bestimmt das Verhalten für den Fall, das die durch **ziel$** spezifizierte Datei schon existiert.
Standard (ohne flags$): vorhandene Dateien (ziel$) immer überschreiben, auch schreibgeschützte und Systemdateien (entspricht dem Flagzeichen "**a**").
Ein oft sinnvoller Wert ist "**am**": Wie Standard, aber bei Problemen eine passende Meldungsbox ("**m**") anzeigen.
Im Abschnitt 8.3 finden Sie eine Auflistung aller Flagzeichen und ihrer Bedeutung sowie Beispiele für ihre Anwendung.

FileCopy kann keine Links kopieren. Die Variable **fileError** wird belegt (d.h. gesetzt oder gelöscht).

Beispiele:
Beispiele unter Verwendung des Flagstrings finden Sie im Abschnitt 8.2.

```
FileCopy "Termine von heute", "Backup der Termine von heute"

!  Kopie in ein entferntes Verzeichnis
datei$ = "Meine Daten"
FileCopy datei$, "F:\\Backup\\Januar\\" + datei$,  "am"
```

FileMove

FileMove verschiebt eine Datei, indem zuerst FileCopy ausgeführt wird und anschließend die Originaldatei gelöscht wird.

Syntax: **FileMove quelle$, ziel$** [, **flags$**]

quelle$, ziel$, flags$: Siehe **FileCopy**.

FileMove kann keine Links verschieben. Die Variable **fileError** wird belegt (d.h. gesetzt oder gelöscht).

Beispiele:

```
FileMove "Termine von heute", "Alte Version\\Termine von heute"
```

```
!  Verschieben aus einem entfernten Verzeichnis
datei$ = "Brief an Willi"
FileMove "F:\\Briefe\\Umzug\\" + datei$,  datei$, "am"
```

Beispiele unter Verwendung des Flagstrings finden Sie im Abschnitt 8.2.

FileDelete

FileDelete löscht eine Datei. Das kann eine DOS oder eine GEOS-Datei oder ein Link sein. Um einen Ordner zu löschen, verwenden Sie bitte **FileDeleteDir**.

Syntax: **FileDelete datei$** [, **flags$**]

datei$	Bezeichnet die zu löschende Datei. Pfadangaben sind zulässig.
flags$	Optional: Einstellung der Reaktion auf Fehler oder bestimmte Situationen. Standard (ohne flags$): Dateien immer löschen, auch schreibgeschützte und Systemdateien (entspricht "a"). Es sind die gleichen Flagzeichen wie bei FileCopy und FileMove zulässig. Ein oft sinnvoller Wert ist "am": Wie Standard, aber bei Problemen eine passende Meldungsbox ("m") anzeigen.

Die Variable **fileError** wird belegt (d.h. gesetzt oder gelöscht). Es ist ein Fehler, wenn die Datei nicht existiert.

Beispiele:
Beispiele unter Verwendung des Flagstrings finden Sie im Abschnitt 8.2.

```
FileDelete  "Meine Daten"
FileDelete  "C:\\TEMP\\Logfile.log" , "am"
```

8.2 Flagzeichen für FileCopy, FileMove und FileDelete

Die Flagzeichen modifizieren das Verhalten von **FileCopy**, **FileMove** und **FileDelete**. Für **FileDelete** kann man so z.B. angeben ob bestimmte Dateien automatisch gelöscht werden oder ob nachgefragt werden soll. Bei **FileCopy** oder **FileMove** kann es passieren, dass das Kopierziel bereits existiert. Hier bestimmen die Flags z.B. ob diese schon vorhandene Datei automatisch gelöscht werden soll oder ob nachgefragt werden soll.

Geben Sie kein Flagzeichen an, so wird der Standard "**a**" genommen.

FileCopy, **FileMove** und **FileDelete** sind standardmäßig (d.h. ohne Angabe spezieller Flagzeichen) so eingestellt, dass sie ihren Dienst "bestmöglich" verrichten. **FileDelete** versucht alle Dateien, auch schreibgeschützte und Systemdateien, zu löschen und **FileMove** bzw. **FileCopy** versuchen, wenn das Kopierziel schon existiert, es zu überschreiben. Für viele Anwendungsfälle ist es daher nicht nötig, sich mit der komplexen Materie der Flagzeichen auseinanderzusetzen.

Konzeption:

Mit Hilfe der Flagzeichen können Sie angeben:

- Welche Dateitypen automatisch überschrieben bzw. gelöscht werden sollen (Flagzeichen "**a**", "**f**" und "**L**"). R-BASIC orientiert sich dabei am Dateityp (siehe Kapitel 6.1). Für schreibgeschützte und Systemdateien gibt es extra Flagzeichen ("**r**" und "**h**"). "**f**" ist eine Abkürzung für "**evgd**".

- Ob nachgefragt werden soll, wenn eine Datei nicht automatisch überschrieben bzw. gelöscht werden soll ("**q**" oder "**u**"), oder ob dies als Fehler gewertet werden soll.

- Ob R-BASIC im Fehlerfall (z.B. Datei nicht gefunden oder Zugriff verweigert) eine Meldung an den Nutzer ausgeben soll ("**m**") oder ob Sie das selbst programmieren wollen (kein "m" angegeben). Der Standard ist, das R-BASIC keine Meldungsbox erzeugt. Es ist oft sinnvoll, "m" anzugeben. Die Variable **fileError** wird in jedem (Fehler-) Fall gesetzt.

- Ob in bestimmten Situationen immer ein Fehler erzeugt werden soll, d.h. die **fileError**-Variable gesetzt werden soll. (Flagzeichen "**o**", "**i**", "**t**")

Sie können den Flagzeichenstring durch Leerzeichen oder Bindestriche übersichtlicher gestalten. Er darf aber nicht länger als 64 Zeichen werden, sonst wird ein Laufzeitfehler erzeugt und das Programm beendet. Die Groß- bzw. Kleinschreibung wird ignoriert.

Allgemeine Flags, die mit allen anderen kombiniert werden können

- **m** **M**essage: Dialogbox anzeigen, wenn ein Fehler auftrat. Unabhängig davon wird die Variable fileError gesetzt.
- **o** Read-**O**nly-Fehler: Erzwingt, dass beim Versuch, eine schreibgeschützte Datei zu überschreiben oder zu löschen immer eine Fehlermeldung erzeugt wird.
- **i** H**i**dden-Fehler: Erzwingt, dass beim Versuch, eine versteckte oder Systemdatei (Attribute FA_HIDDEN oder FA_SYSTEM) zu überschreiben oder zu löschen immer eine Fehlermeldung erzeugt wird.
- **t** **T**ype-Fehler (nur FileMove und FileCopy): Erzwingt, dass beim Versuch, eine DOS- durch eine GEOS-Datei (oder umgekehrt) zu überschreiben, immer eine Fehlermeldung erzeugt wird.

Das Flags "**o**" "**i**" "**t**" haben Vorrang vor allen anderen Flags. Die Variable fileError wird gesetzt und falls das Flag "**m**" angegeben wurde wird eine entsprechende Dialogbox angezeigt.

Flags für Dateien, die gelöscht bzw. überschrieben werden können

- **f** **F**iles (=Dateien) Normale Dateien sollen ohne Nachfragen überschrieben bzw. gelöscht werden.
  Das schließt **nicht** ein: Links, schreibgeschützte, versteckte und Systemdateien.
  Das Flag "**f**" ist eine Abkürzung für "**e v g d**". Statt "f" anzugeben kann man einzelne Dateitypen angeben:
  - **e** Executable: Applikationen bzw. Libraries
  - **v** VM-Dateien
  - **g** Geos-Daten-Dateien
  - **d** DOS-Dateien
- **L** **L**inks sollen ohne Nachfragen überschrieben bzw. gelöscht werden.
- **r** **R**ead-only: Schreibgeschützte Dateien sollen ohne Nachfragen überschrieben bzw. gelöscht werden.
- **h** **H**idden: Versteckte Dateien (Attribute FA_HIDDEN oder FA_SYSTEM) sollen ohne Nachfragen überschrieben bzw. gelöscht werden.
- **a** **A**lle Dateien: "**a**" ist eine Abkürzung für "**f L r h**"

Behandlung von Dateien, die überschrieben oder gelöscht werden solle, aber **nicht** durch die Dateiflags oben **erfasst** sind

q oder **u** **Qu**estion: R-BASIC fragt nach, ob die Datei gelöscht / überschreiben werden soll. Der Unterschied ist:

- **q** Antwortet der Nutzer mit "Nein" so wird Variable fileError auf –1 (Abbruch durch Nutzer) gesetzt.
- **u** Antwortet der Nutzer mit "Nein" so wird Variable fileError gelöscht (Wert Null, OK).

Wird weder "**q**" noch "**u**" angegeben, so wird die Datei nicht gelöscht / überschrieben und die Variable fileError wird auf +5 (Zugriff verweigert) gesetzt. Das ermöglicht Ihnen, ein flexibles Fehlerhandling zu implementieren.

Beispiele für sinnvolle Flagzeichenstrings:

"**a** " Dies ist der **Standard** für FileCopy, FileMove und FileDelete.
Alle Dateien und Links löschen / überschreiben, aber bei Fehler keine Meldung machen (nur fileError setzen). Ihr Programm übernimmt die Fehlerbehandlung selbst.

"**a m**" Alle gefunden Dateien und Links werden überschrieben bzw. gelöscht (auch schreibgeschützte und Systemdateien).
Meldung machen bei Fehler ("m"), z.B. Datei nicht gefunden.

"" (leerer String) Niemals Dateien löschen/überschreiben (sinnvoll für FileCopy und FileMove). fileError wird auf +5 (Zugriff verweigert) gesetzt, falls das Ziel schon existiert. Ihr Programm übernimmt die Fehlerbehandlung selbst.

"**f Lq m**" Alle Dateien ("**f**") und Links ("**L**") löschen / überschreiben, aber Nachfragen (Question "**q**") bei schreibgeschützten und Systemdateien (diese sind von "**f**" nicht abgedeckt).
Meldung machen bei Fehler ("m").

"**f Lrq m**" Wie "**f Lq m**" (siehe letztes Beispiel), aber schreibgeschützte Dateien ("**r**") automatisch überschreiben und nur bei Systemdateien nachfragen.

"**f Lq m i**" Alle Dateien ("**f**") und Links ("**L**") löschen / überschreiben, Nachfragen (Question "**q**") bei schreibgeschützten Dateien, aber bei Systemdateien ("**i**") immer eine Fehler erzeugen (**fileError** setzen, aber nicht nachfragen)
Meldung machen bei Problemen ("**m**").

"**a o i t m**" Alle Dateien und Links löschen / überschreiben, außer bei schreibgeschützten ("**o**"), versteckten bzw. Systemdateien ("**i**") und bei Typ-Fehlern ("**t**"): diese nicht löschen/überschreiben, Meldung machen ("**m**") und die fileError-Variable setzen.

Codebeispiele unter Verwendung der Flagzeichen

Wenn es zu einem Problem kommt (z.B. Datei nicht gefunden) wird immer die Variable **fileError** gesetzt. Mit dem Flagzeichen "**m**" erzeugt R-BASIC zusätzlich eine Dialogbox, die den Fehler beschreibt.

Vorhandene Datei immer löschen, und bei Fehler eine Dialogbox ausgeben.

```
FileCopy "Meine Daten" ,  "A:\\Meine Daten" ,"am"
FileDelete "A:\\Meine Daten" ,"am"
```

Der Standard ist, bei Problemen keine Meldungsbox auszugeben. Das entspricht dem Flagzeichenstring "**a**". Das Programm sollte dann eventuelle Fehler selbst behandeln.

```
FileDelete "A:\\Meine Daten"        ' Entspricht "a"
IF  fileError THEN .....
```

Immer nachfragen wenn, wenn das Ziel schon existiert. Mit "m": R-BASIC meldet, wenn sich das Ziel nicht überschrieben lässt (ohne "m" wird nur fileError gesetzt)
Antwortet der Nutzer auf Nachfrage mit "Nein", wird fileError auf –1 (Abbruch durch Nutzer) gesetzt.

```
FileMove  quelle$, ziel$, "q"
FileMove  quelle$, ziel$, "qm"
```

Wie letztes Beispiel, aber bei "Nein" wird fileError auf Null gesetzt.

```
FileMove   quelle$, ziel$, "u"
FileMove   quelle$, ziel$, "um"
```

Normale Dateien überschrieben ("**f**"), bei schreibgeschützten und Systemdateien nachfragen. Mit "**m**": R-BASIC erzeugt eine Fehlerbox, wenn es ein Problem gab. Ohne "m": Ihr Programm ist verantwortlich indem es die Variable fileError abfragt.

```
FileCopy   quelle$, ziel$, "fqm"

FileCopy   quelle$, ziel$, "fq"
IF  fileError  THEN
  MsgBox "Fehler beim Kopieren von "+datei$ + " :\r" +
          ErrorText$(fileError)
END IF
```

Normale Dateien überschreiben bzw. löschen ("**f**"), bei schreibgeschützten nachfragen ("**q**") und bei Systemdateien immer fileError setzen ("i"). Die Leerzeichen zwischen den Flagzeichen sind zulässig und verbessern die Übersicht.

```
FileCopy   quelle$, ziel$, "f q i m"
FileDelete ziel$, "f q i m"
```

8.3 Arbeit mit Dateinamen

GEOS-Dateien und Ordner haben neben ihrem langen, unter GEOS sichtbaren Namen noch einen DOS-Namen. Normalerweise wird der DOS-Name vom System automatisch vergeben. R-BASIC hat Zugriff sowohl auf den GEOS- als auch auf den DOS-Namen von Dateien und verfügt über die einzigartige Fähigkeit, den DOS-Namen bewusst zu manipulieren.

FileRename

Ändert den Namen einer DOS- oder GEOS-Datei oder eines Ordners. Der DOS-Name einer GEOS-Datei wird dabei vom System ebenfalls geändert.

Syntax: **FileRename oldName$, newName$** [, **flags$**]

oldName$: Alter Dateiname. oldName$ darf einen kompletten Pfad enthalten. Bei GEOS-Dateien bzw. Ordnern ist die Groß- Kleinschreibung zu beachten.
newName$: Neuer Dateiname. newName$ darf keinen Pfad enthalten. Für DOS-Dateien muss newName$ der Konvention 8.3 entsprechen.
flags$: (optional): Zeichenkette, bestehend aus dem Buchstaben "m" (für Message), die festlegt, ob im Fehlerfall (z.B. Datei nicht gefunden, neuer Name ungültig) eine Meldungsbox angezeigt wird.
Wird flags$ nicht angegeben, gibt es keine Meldungsbox.

Auch schreibgeschützte, System- oder versteckte Dateien sowie GEOS-Links können umbenannt werden. Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiele:

```
' Einfaches umbenennen
FileRename "E:\\Dateien\\info.txt", "info.bak"
```

```
' Umbenennen mit automatischer Fehlermeldung durch R-BASIC
FileRename "E:\\Dateien\\info.txt", "info.bak", "m"
```

```
' Umbenennen mit eigener Fehlermeldung
FileRename "E:\\Dateien\\info.txt", "info.bak"
IF   fileError  THEN MsgBox("Die Datei konnte nicht umbenannt
                          werden\rFehler: " +
                          ErrorText$(fileError))
```

FileGetDosName$

Liefert den DOS-Namen einer Datei oder eines Ordners.

Syntax: **FileGetDosName$** (**geosName$**)

geosName$: Der GEOS-Name der Datei.
geosName$ darf auch eine DOS-Datei bezeichnen.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiel:

```
DIM name$
name$ = "Write unbenannt"
Print  name$,  FileGetDosName$ ( name$)
```

FileSetDosName

Ändert den DOS-Namen einer GEOS-Datei oder eines Ordners. Der lange GEOS-Name wird nicht geändert.

Syntax: **FileSetDosName** **oldName$**, **newName$** [, **flags$**]

oldName$, newName$, flags$: siehe **FileRename**

Die Systemvariable **fileError** wird gesetzt oder gelöscht. Auch schreibgeschützte, System- oder versteckte Dateien können umbenannt werden.

Beispiele:

```
' Einfaches umbenennen
FileSetDosName "E:\\Dateien\\Draw Beispiel", "DRAW.777"
```

```
' Umbenennen mit automatischer Fehlermeldung durch R-BASIC
FileSetDosName "E:\\Dateien\\Draw Beispiel", "DRAW.777", "m"
```

```
' Umbenennen mit eigener Fehlermeldung
FileSetDosName "E:\\Dateien\\Draw Beispiel", "DRAW.777"
IF  fileError  THEN MsgBox("Der DOS-Name der Datei konnte nicht
                           geändert werden\rFehler: " +
                           ErrorText$(fileError))
```

8.4 Suchen nach Dateien

R-BASIC unterstützt die Suche nach Dateien und Ordnern mit den Befehlen **FileFindFirst$**, **FileFindNext$** und **FileFindDone**. FileFindFirst$ durchsucht den Ordner und initialisiert ein Handle, das von FileFindNext$ und FileFindDone verwendet wird. FileFindFirst$ liefert den ersten Datei- bzw. Ordnernamen. Die nächsten Datei- bzw. Ordnernamen werden von FileFindNext$ geliefert.
FileFindDone gibt nach getaner Arbeit das Handle wieder frei.
FileFindFirst$, FileFindNext$ und FileFindDone beeinflussen fileError nicht.

FileFindFirst$

Sucht den ersten Datei- bzw. Ordnernamen. Initialisiert ein Handle, dass von FileFindNext$ und FileFindDone verwendet wird.

Syntax: **<name$>** = **FileFindFirst$** (**<han>** [, **mask$** [, **flags$** [, **<token>**]]])

<han>: Variable vom Typ Handle. Ausdrücke (z.B. Funktionsaufrufe) sind nicht zulässig.

mask$: (optional) Dateimaske, die auf die zu findende Datei passen muss.
Default: "*" (= alle Dateien und Ordner finden)
Es gelten die GEOS-Namens-Konventionen:
* (Sternchen): beliebige Anzahl (oder Null) Zeichen oder Ziffern
? Genau ein Zeichen oder eine Ziffer
: und \ sind nicht zulässig
Groß- bzw. Kleinschreibung und Leerzeichen werden berücksichtigt.
Für DOS-Dateien sind Großbuschstaben anzugeben.
Die Maske darf keinen Pfadanteil enthalten!
Beispiele: "*a*" findet alle Dateien, deren Name ein 'a' enthält
"X*" findet alle Dateien, deren Name mit 'X' beginnt
"X*e" Der Name muss mit 'X' beginnen und auf "e" enden.

flags$: (optional) Zeichenkette, die bestimmt, welche Dateitypen gefunden werden sollen. Zulässig sind:
e Executable: Applikationen bzw. Libraries
g oder v Geos-Daten-Dateien bzw. VM-Dateien
Eine Unterscheidung zwischen beiden ist nicht möglich
d DOS-Dateien
f oder o Folders (Ordner)
a Alle Dateien und Ordner. Abkürzung für "e g d f"
L Links
Wird flags$ nicht angegeben wird "a L" angenommen (alles finden).

<token>: (optional) GeodeToken der zu findenden Datei. Wird "token" nicht angegeben, gibt es keine Einschränkung. Um "token" angeben zu können, muss man "flags$" angeben und "flags$" muss GEOS-Dateien einschließen (e, g bzw. a).
Hinweis: Ordner werden immer gefunden (wenn das Flag "f" angegeben ist), auch wenn sie nicht das entsprechende Token haben.

Die Systemvariable fileError wird nicht beeinflusst. Ist im aktuellen Ordner keine Datei oder Unter-Ordner enthalten, auf welche die Suchkriterien passen, wird ein Leerstring (Länge Null) zurückgegeben. Auch in diesem Fall muss FileFindDone gerufen werden!

FileFindNext$

Sucht den nächsten Datei- bzw. Ordnernamen. Ist keine weitere Datei oder Ordner verfügbar, wird ein Leerstring (Länge Null) zurückgegeben.

Syntax: **<name$>** = **FileFindNext$** (**<han>**)
<han>: Variable (oder Ausdruck) vom Typ Handle. han muss von FileFindFirst$ initialisiert worden sein.

Die Systemvariable fileError wird nicht beeinflusst.

FileFindDone

Gibt das von FileFindFirst$ initialisierte Handle frei, indem die dahinterliegenden Datenstrukturen und Speicherbereiche freigeben werden. Dieser Schritt ist sehr wichtig, da Speicher im GEOS-System knapp ist.

Syntax: FileFindDone **<han>**
<han>: Variable (oder Ausdruck) vom Typ Handle. han muss von FileFindFirst$ initialisiert worden sein.

Die Systemvariable fileError wird nicht beeinflusst.

Beispiel 1: Dateien und Ordner auflisten:

```
DIM    han AS   HANDLE
DIM    name$

name$ = FileFindFirst$ ( han )   ! Handle initialisieren
WHILE  ( name$ <> "" )           ! Vergleich auf Leerstring
  Print name$,                   ! Komma am Ende
                                 ! --> tabuliert
  IF FileType(name$) = GFT_DIRECTORY THEN
        Print "<DIR>"
     ELSE
        Print  FileSize(name$); " Bytes"
     END IF
  name$ = FileFindNext$ ( han )      ! Handle benutzen
  WEND

FileFindDone ( han )        ! Handle freigeben.
                        ! Die Klammern sind optional.
! Die von han referenzierten Datenstrukturen werden
! freigegeben.
! Die in han gespeicherten Werte sind jetzt ungültig.
```

Beispiel 2: Alle Write-Dateien im Geos Top-Folder auflisten:

```
DIM han  AS   HANDLE
DIM name$
DIM token AS GeodeToken

  token.manufid = 0
  token.tokenChars = "WDAT"

  CLS
  SetStandardPath  SP_TOP

  name$ = FileFindFirst$ ( han, "*", "aL", token )  !
  WHILE  ( name$ <> "" )
    if FileType(name$) <> GFT_DIRECTORY THEN Print name$
    name$ = FileFindNext$ ( han )          ! Handle benutzen
  WEND

  FileFindDone ( han )            ! free handle
```

(Leerseite)

9 Arbeit mit Dateien

9.1 Überblick zur Dateiarbeit

Für R-BASIC ist jede Datei zunächst eine einfache Abfolge von Bytes. Um auf die Daten in einer Datei zugreifen zu können, müssen Sie sie zuerst "öffnen" und nach Gebrauch wieder "schließen". Während die Datei geöffnet ist, erfolgt der Zugriff auf eine Datei über eine Variablen vom Typ **FILE**, die von **FileOpen** bzw. **FileCreate** geliefert wird. GEOS verwaltet außerdem einen "Dateizeiger", der die Position bestimmt, an der Daten gelesen oder geschrieben werden.

R-BASIC verfügt dabei über einige Möglichkeiten, die den meisten Programmiersprachen fehlen, wie z.B. das direkte Einfügen von Daten, ohne die darauf folgenden Daten zu überschreiben.

Hier finden Sie eine Übersicht über die Befehle zum Lesen aus und Schreiben von Daten in Dateien, die in diesem Kapitel behandelt werden.

Dateiattribute (Abschnitt 9.2)

DOS-Attribute

FileGetAttrs, FileSetAttrs
: Lesen und setzen die "Standard-Attribute" wie Archiv, schreibgeschützt usw.

FileGetTime, FileSetTime
: Lesen und setzen das Datum der letzten Änderung.

FileSize
: liefert die aktuelle Dateigröße.

GEOS-Attribute

Zusätzlich zu den DOS-Attributen haben GEOS-Dateien weitere Attribute.
R-BASIC unterstützt die folgenden Attribute:

Token
: Dieses Attribut bestimmt das "Icon", dass im GeoManager für diese Datei angezeigt wird.
Befehle: **FileSetToken, FileGetToken**

Creator
: Erzeuger. Dieses Attribut enthält das Token des Programms, das die Datei angelegt hat.
Befehle: **FileSetCreator, FileGetCreator**
Token und Creator werden jeweils in einer **GeodeToken** Struktur gespeichert.

CreationTime
: Datum und Zeit, zu der die Datei angelegt wurde.
Befehle: **FileSetCreationTime, FileGetCreationTime**

Usernotes
: Die Benutzer-Notizen. Ein String mit bis zu 99 Zeichen.
Befehle: **FileSetUsernotes, FileGetUsernotes**

Release
: Dieses Attribut entspricht der Versionsnummer der Datei. R-BASIC und auch der Uni-Installer entscheiden darüber, ob eine Datei neuer ist als eine andere gleichen Namens.
Befehle: **FileSetRelease, FileGetRelease**

Protocol
: Dieses Attribut beschreibt intern die "Fähigkeiten" einer Datei.
Befehle: **FileSetProtocol, FileGetProtocol**

Anlegen, Öffnen und Schließen von Dateien (Abschnitt 9.3)

FileOpen Öffnet eine vorhandene Datei
FileCreate Legt eine neue Datei an oder öffnet eine vorhandene Datei
FileClose Schließt eine Datei

Lesen und Schreiben von Daten - Binärdateien (Abschnitt 9.4)

FileRead, **FileWrite** und **FileInsert** sind Universalroutinen. Sie arbeiten mit allen Dateien und allen Typen von Variablen, einschließlich Strings und Strukturen zusammen. Für die Arbeit mit Textdateien (lesen und schreiben von Textzeilen) gibt es zusätzlich dazu spezialisierte Routinen (Abschnitt 9.5).

FileRead Liest eine bestimmte Anzahl von Bytes aus einer Datei.
FileWrite Schreibt eine bestimmte Anzahl von Bytes in eine Datei, indem vorhandene Daten überschrieben werden. Bei Bedarf werden die Daten angehängt, d.h. die Datei wird verlängert.
FileInsert Fügt eine bestimmte Anzahl von Bytes in eine Datei ein, ohne das vorhandene Daten überschrieben werden. Die Datei wird dadurch automatisch verlängert. Dieser Befehl ist eine Besonderheit von R-BASIC, die meisten Programmiersprachen kennen ihn nicht.

Lesen und Schreiben von Daten - Textdateien (Abschnitt 9.5)

Diese Routinen sind auf Textzeilen, die durch ein **Zeilenendezeichen** abgeschlossen sind, spezialisiert. Das trifft für normale Textdateien zu. Zeilenendezeichen sind die ASCII-Codes 13 (Wagenrücklauf, Carriage return, CR) bzw. 10 (Zeilenvorschub, LineFeed, LF) oder eine Kombination davon, üblicherweise die Folge 13, 10 (CRLF).

FileReadLine$ Liest eine Textzeile aus einer Datei.
FileWriteLine Schreibt eine Textzeile, indem vorhandene Daten überschrieben werden. Bei Bedarf wird die Textzeile angehängt, d.h. die Datei wird verlängert.
FileInsertLine Schiebt eine Textzeile in die Datei ein. Die Datei wird dadurch verlängert.
FileReplaceLine Ersetzt eine Textzeile in einer Datei durch eine andere.

Sonstige Funktionen (Abschnitt 9.6)

FileGetPos Liest den aktuellen Dateizeiger.
FileSetPos Setzt den Dateizeiger.
FileResize Ändert die Größe einer Datei, indem Daten eingefügt oder gelöscht werden.
FileTruncate Schneidet die Datei ab.
FileCommit Stellt sicher, dass eventuell vom System gepufferte Daten sofort, d.h. schon vor einem FileClose auf die Platte geschrieben werden.

9.2 Dateiattribute

Neben dem Namen besitzt jede Datei eine eine Reihe von zusätzlichen Eigenschaften, die sogenannten "Attribute". Es gibt zwei Gruppen von Attributen, die "Standard-Attribute", einschließlich Dateigröße und dem Datum der letzten Änderung, die bereits aus alten DOS-Tagen stammen, und die "GEOS-Attribute", die nur für GEOS-Dateien vorhanden sind.

Die meisten der in diesem Kapitel besprochenen Funktionen können sowohl mit offenen Dateien (referenziert über eine FILE-Variable) als auch mit geschlossenen Dateien (referenziert über ihren Namen) umgehen.

Die Standard-Attribute

Die Standardattribute existieren für JEDE Datei und sind als einzelne Bits in einem Byte definiert. Sie können mit **FileGetAttrs** gelesen und mit **FileSetAttrs** gesetzt werden. Es sind die folgenden Attribute definiert, die Zahlen in der ersten Spalte sind die Bit-Nummer und der dazugehörige Wert.

Bit (Wert)	BASIC-Konstante	Bedeutung
Bit 0 (1)	FA_READ_ONLY	Read-Only. Die Datei ist schreibgeschützt.
Bit 1 (2)	FA_HIDDEN	Hidden. Die Datei ist versteckt.
Bit 2 (4)	FA_SYSTEM	System. Es ist eine wichtige Systemdatei.
Bit 3 (8)	FA_VOLUME	Volume. Es ist keine Datei, sondern der Eintrag für die Datenträgerbezeichnung.
Bit 4 (16)	FA_SUBDIR	Subdir. Es ist keine Datei, sondern ein Verzeichnis (= Ordner).
Bit 5 (32)	FA_ARCHIVE	Archive. Die Datei wurde geändert. Dieses Bit wird bei jedem Schreibzugriff auf die Datei wieder gesetzt, so das Backup-Programm daran erkennen können, ob die Datei gesichert werden muss. Sie setzten dieses Bit nach dem Sicherungsprozess zurück.
Geos verwendet weiterhin:		
Bit 6 (64)	FA_LINK	Link. Dies ist kein Standardattribut. Ist dieses Bit gesetzt, handelt es sich nicht um eine echte Datei, sondern einen GEOS-internen Link auf eine Datei.

FileGetAttrs

Liefert die DOS-Standardattribute einer Datei. Die Attribute sind einzelne Bits und können mit AND bzw. OR - Verknüpfungen abgefragt werden.

Syntax: **<numVar>** = **FileGetAttrs** (**fileName$**)
oder: **<numVar>** = **FileGetAttrs** (**<fh>**)

fileName$: Name der Datei. Pfadangaben im Namen sind zulässig.
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiel:

```
DIM     attrs AS word
attrs = FileGetAttrs ( "info.txt" )
IF  attrs AND FA_READ_ONLY   THEN   Print "schreibgeschützt"
```

FileSetAttrs

Setzt die DOS-Standardattribute einer Datei. Die Attribute sind einzelne Bits und können mit AND bzw. OR verknüpft werden.

Syntax: **FileSetAttrs** **fileName$** , **attrs**

fileName$: Name der Datei. Pfadangaben im Namen sind zulässig.
attrs: Neue Attribute

Die Systemvariable **fileError** wird gesetzt oder gelöscht. **FileSetAttrs** kann nicht mit einer offenen Datei verwendet werden.

Beispiele:

```
! Setzen von FA_SYSTEM, löschen aller anderen Attribute
FileSetAttrs  "liste.txt" , FA_SYSTEM

! Setzen von FA_SYSTEM unter Beibehaltung der anderen Attribute
DIM     attrs
attrs = FileGetAttrs ( "liste.txt" )
FileSetAttrs  "liste.txt" , attrs   OR   FA_SYSTEM
```

FileGetTime

Liefert das Datum und die Uhrzeit der letzten Änderung einer Datei. Das Betriebssystem setzt diesen Wert jedes Mal, wenn der Dateiinhalt geändert wird.

Syntax: **<time>** = **FileGetTime** (**fileName$**)
oder: **<time>** = **FileGetTime** (**<fh>**)

fileName$: Name der Datei. Pfadangaben im Namen sind zulässig.
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.
<time>: Eine Variable vom Typ **DateAndTime**

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiel:

```
DIM  time   AS DateAndTime
time  = FileGetTime  "info.txt"
Print "Geändert am: "; FormatDate$(time);
Print "um: "; FormatTime$(time)
```

FileSetTime

Verändert das Datum und die Uhrzeit der letzen Änderung einer Datei.
Achtung! Sie überschreiben hiermit den Wert, den das Betriebssystem automatisch vergibt. Wird die Datei anschließend nochmals geändert, überschreibt das Betriebssystem den Wert erneut.

Syntax: **FileSetTime** **fileName$** , **<time>**
oder: **FileSetTime** **<fh>** , **<time>**

fileName$: Name der Datei. Pfadangaben im Namen sind zulässig.
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.
<time>: Eine Variable vom Typ DateAndTime

Fehlerbedingung: Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiel:

```
DIM time AS DateAndTime
time  = FileGetTime  ( "info.txt" )
time.year = 2001
FileSetTime   "info.txt", time
```

FileSize

Liefert die aktuelle Größe der Datei in Bytes. Der zurückgegebene Wert liegt im Bereich von 0 bis 4294967295, da Dateien maximal 4 GByte groß werden können.

Syntax: **<numVar>** = **FileSize** (**fileName$**)
oder: **<numVar>** = **FileSize** (**<fh>**)

fileName$: Name der Datei. Pfadangaben im Namen sind zulässig.
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

GEOS-Attribute

Zusätzlich zu den Standardattributen haben GEOS-Dateien weitere Attribute. R-BASIC unterstützt die wichtigen Attribute: **Token** (Anzeige-Icon), **Creator** (Icon des zugehörigen Programms), **CreationTime** (Datum der Dateierstellung), **UserNotes** (Benutzernotizen), **Release** (Versionsnummer) und **Protocol**. Diese werden im Dateikopf, den ersten 256 Byte der Datei gespeichert. Bei Verzeichnissen befinden sich diese Attribute in der @dirname.000 - Datei.

CreationTime wird ein einer **DateAndTime** Struktur gespeichert.

```
STRUCT DateAndTime
   year     AS  WORD     ' Jahr      (z.B. 2014)
   month    AS  WORD     ' Monat     (1...12)
   day      AS  WORD     ' Tag       (1 ... 31)
   hour     AS  WORD     ' Stunde    (0 ... 23)
   minute   AS  WORD     ' Minute    (0 ... 59)
   second   AS  WORD     ' Sekunde   (0 ... 59)
 END  STRUCT
```

UserNotes ist ein String mit bis zu 99 Zeichen.

Token und **Creator** werden in einer Struktur gespeichert, die **GeodeToken** heißt und folgendermaßen definiert ist. Das Bild dazu befindet sich in der TokenDB-Datei.

```
 STRUCT GeodeToken
   manufid  AS   WORD
            ' Manufacturer ID (Hersteller-Identifikation)
   tokenChars   AS   STRING(4)
 END  STRUCT
```

Eine fehlerhafte Belegung der Werte für Token oder Creator kann die Arbeit mit der Datei unmöglich machen. Setzen Sie in diesem Fall einfach wieder den korrekten Wert.

Release und **Protocol** werden in einer Struktur gespeichert die **ReleaseNumber** heißt und folgendermaßen definiert ist. R-BASIC verwendet die ersten beiden Felder dieser Struktur auch für die Protokollnummer.

```
STRUCT ReleaseNumber          ' Bedeutung der Felder:
  rnMajor as WORD             ' große, meist inkompatible Neuerungen
  rnMinor as WORD             ' kleinere, kompatible Neuerungen
  rnChange AS WORD            ' interne Änderungen
  rnEngineering as WORD       ' kleine interne Änderungen
End STRUCT
```

Tritt ein Fehler auf, z.B. weil die Datei nicht gefunden wird oder weil das Attribut nicht unterstützt wird, (weil es eine DOS-Datei ist), wird die Systemvariable **fileError** gesetzt. Im Erfolgsfall wird die **fileError**-Variable zurückgesetzt (d.h. mit Null belegt)

Die Bedeutung der Parameter der folgenden Übersicht und Beispiele sind weiter unten zu finden.

FileGetToken, FileSetToken

Liest bzw. setzt das "Token" der Datei.

Syntax: **<token>** = **FileGetToken** (f**fileName$**) ' Auslesen des Token
oder: **<token>** = **FileGetToken** (**<fh>**)

Syntax: **FileSetToken** **fileName$**, **<token>** ' Setzen des Token
oder: **FileSetToken** fh, **<token>**

FileGetCreator, FileSetCreator

Liest (FileGetCreator) oder setzt (FileSetCreator) das Creator-Token.

Syntax: **<token>** = **FileGetCreator** (**fileName$**) ' Auslesen des Creator-Token
oder: **<token>** = **FileGetCreator** (**<fh>**)

Syntax: **FileSetCreator** **fileName$**, **<token>** ' Setzen des Creator-Token
oder: **FileSetCreator** **<fh>** , **<token>**

FileGetCreationTime, FileSetCreationTime

Liest oder verändert das Datum, an dem die Datei angelegt wurde.

Syntax: **<time>** = **FileGetCreationTime** (**fileName$**) ' Auslesen der Erstellungszeit
oder: **<time>** = **FileGetCreationTime** (**<fh>**)

Syntax: **FileSetCreationTime** **fileName$**, **<time>** ' Setzen der Erstellungszeit
oder: **FileSetCreationTime** **<fh>** , **<time>**

FileGetUsernotes, FileSetUsernotes

Liest oder verändert die Benutzernotizen der Datei.

Syntax: **notes$** = **FileGetUsernotes** (**fileName$**) ' Auslesen der Benutzernotizen
oder: **notes$** = **FileGetUsernotes** (**<fh>**)

Syntax: **FileSetUsernotes** **fileName$**, **notes$** ' Setzen der Benutzernotizen
oder: **FileSetUsernotes** **<fh>**, **notes$**

FileGetRelease, FileSetRelease

Liest oder verändert die Releasenummer der Datei. Das GEOS System verwendet die Releasenummer zum Versionscheck und zu Informationszwecken. R-BASIC und der Uni-Installer verwenden die Releasenummer um zu entscheiden ob eine Datei neuer ist als eine andere Datei mit gleichem Namen. Der R-BASIC Compiler setzt die ersten drei Felder der Releasenummer von Launcher und BIN-Datei entsprechend der Versionsnummer des Programms bzw. der Library (siehe Befehl Version$).
Bei R-BASIC Libraries wird die Releasenummer (und damit die Versionsnummer der Library) verwendet um zu entscheiden, ob ein BASIC Programm mit dieser Library zusammenarbeiten kann.
Um zu entscheiden, ob die R-BASIC IDE die Datei öffnen und bearbeiten kann, wird die Protokollnummer, nicht die Releasenummer, benutzt.

Syntax: **<release>** = **FileGetRelease** (**fileName$**)
oder: **<release>** = **FileGetRelease** (**<fh>**) ' Auslesen der Release-Nummer

Syntax: **FileSetRelease** **fileName$**, **<release>**
oder: **FileSetRelease** **<fh>** , **<release>** ' Setzen der Release-Nummer

FileGetProtocol, FileSetProtocol

Liest oder verändert die Protokollnummer der Datei. Das GEOS System verwendet die Protokollnummer um zu prüfen, ob Programme, Dateien und Libraries kompatibel zueinander sind. R-BASIC definiert keine eigene Struktur für die Protokollnummer, sondern verwendet die Felder rnMajor und rnMinor der ReleaseNumber Struktur. **Vorsicht!** Änderungen der Protokollnummer können dazu führen, dass Programme nicht mehr funktionieren oder Dateien nicht mehr gelesen werden können.

Syntax: **<protocol>** = **FileGetProtocol** (**fileName$**)
oder: **<protocol>** = **FileGetProtocol** (**<fh>**) ' Auslesen der Protocol-Nummer

Syntax: **FileSetProtocol** **fileName$**, **<protocol>**
oder: **FileSetProtocol** **<fh>** , **<protocol>** ' Setzen der Protocol-Nummer

Angaben zu den Parametern und Rückgabewerten:

<token>:	Eine Variable vom Typ "GeodeToken" Die Set- Routinen akzeptieren auch Funktionen, die eine GeodeToken-Struktur zurückgeben
fileName$:	Name der Datei Stringausdrücke, auch mit Pfadangaben im Namen, sind zulässig.
<fh>:	Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.
<time>:	Eine Variable vom Typ "DateAndTime". Die Set- Routinen akzeptieren auch Funktionen, die eine DateAndTime-Struktur zurückgeben.
notes$:	Ein String mit bis zu 99 Zeichen. Für FileSetUsernotes sind Stringausdrücke zulässig. Ist der String zu lang tritt ein Laufzeitfehler auf und das Programm wird beendet.
<release> <protocol>:	Eine Variable vom Typ "ReleaseNumber". Für die "Protocol"-Funktionen werden nur die Felder rnMajor und rnMinor verwendet. Die Set- Routinen akzeptieren auch Funktionen, die eine ReleaseNumber-Struktur zurückgeben.

Beispiel:

```
DIM token    AS GeodeToken
DIM time     AS DateAndTime
DIM release  AS ReleaseNumber
DIM text$

  SetStandardPath  SP_SYSTEM

  token = FileGetToken "geos.geo"
  Print "Token: \"";token.tokenchars;"\","; token.manufid
                          ' manufid = Manufacturer ID

  token = FileGetCreator "geos.geo"
  Print "Creator: \"";token.tokenchars;"\","; token.manufid

  time = FileGetCreationTime "geos.geo"
  Print "Datum: "; FormatDate$(time)
  Print "Zeit: "; FormatTime$(time)

  release = FileGetRelease "geos.geo"
  text$ = "Version: "+ Trim$(Str$(release.rnMajor))
  text$ = text$ + "."+ Trim$(Str$(release.rnMinor))
  text$ = text$ + "  "+ Trim$(Str$(release.rnChange))
  text$ = text$ + "-"+ Trim$(Str$(release.rnEngineering))
  Print text$

  release = FileGetProtocol "geos.geo"
  text$ = "Protocol: "+ Trim$(Str$(release.rnMajor))
  text$ = text$ + "."+ Trim$(Str$(release.rnMinor))
  Print text$
```

9.3 Anlegen, Öffnen und Schließen von Dateien

Um mit einer Datei arbeiten zu können, müssen Sie sie zuerst "öffnen" (was beim Anlegen automatisch geschieht) und nach Gebrauch wieder "schließen". Dabei legen Sie fest, ob Sie Daten in die Datei schreiben (**write**), aus ihr lesen (**read**) oder beides (read/write) wollen (Parameter 'accessFlags$'). Read/Write ist der Standard. Da GEOS ein MultiThread-System ist, kann es passieren, dass andere Programme **gleichzeitig** auf diese Datei zugreifen wollen. Deswegen müssen Sie festlegen ob und wie andere Programme gleichzeitig auf Ihre Datei zugreifen dürfen (**read, write, read/write** bzw. **gar nicht**). Der Standard ist, keinerlei Zugriff zu erlauben (Parameter 'alienFlags$'). Sie sollten davon nur abweichen, wenn es unbedingt erforderlich ist, da das System in diesem Fall viele Daten zwischenspeichern muss, was den Systemspeicher sehr belasten kann.

FileCreate

Legt eine Datei auf dem Datenträger an und öffnet sie. Existiert die Datei schon, kann sie auch verwendet (Daten bleiben erhalten) oder verworfen werden.

Syntax: **<fh>** = **FileCreate** **fileName$** [, **accessFlags$** [, **alienFlags$**]]

<fh>:	Variable vom Typ FILE. fh enthält dann eine Referenz auf die Datei und wird für alle anderen Dateioperationen benötigt.
fileName$:	Name der Datei.
accessFlags$:	Zugriffsflags (optional): Zeichenkette, bestehend aus einer Kombination der Buchstaben "otn g rw x", die den Zugriff auf die Datei festlegen. Der Standard (accessFlags$ nicht angegeben) ist "n rw": Neue Datei zum Lesen und Schreiben anlegen.
alienFlags$:	Fremdzugriffsflags (optional): Zeichenkette, bestehend aus einer Kombination der Buchstaben "rw", die den Zugriff auf die Datei festlegen. Der Standard (alienFlags$ nicht angegeben) ist "": Keine Fremdzugriffe erlaubt.

Der Dateizeiger ist nach dem Anlegen einer Datei immer auf die Position 0 gesetzt, auch wenn sie schon Daten enthält. Wollen Sie Daten anhängen, müssen Sie ihn zunächst mit der Anweisung **FileSetPos** ans Dateiende setzen.

Die Systemvariable **fileError** wird gesetzt oder gelöscht. Beispiele und eine ausführliche Beschreibung der Flagzeichen finden Sie weiter unten.

Mögliche Fehlerbedingungen für FileCreate sind unter anderem:

- Die Datei existiert, aber es wurde ’n’ (nur neu anlegen) angegeben.
- Die Datei existiert als DOS-Datei, aber es wurde ’g’ (GEOS Daten-Datei anlegen) angegeben.
- Die Datei existiert und es wurde ’o’ (open) bzw. ’t’ (truncate, abschneiden) angegeben, aber sie ist von einem anderen Programm geöffnet, und dieses verweigert den Zugriff.

FileOpen

Öffnet eine Datei. Die Datei muss schon auf dem Datenträger vorhanden sein.

Syntax: **<fh>** = **FileOpen** **fileName$** [, **accessFlags$** [, **alienFlags$**]]

<fh> Variable vom Typ FILE. fh enthält dann eine Referenz auf die Datei und wird für alle anderen Dateioperationen benötigt.

fileName$: Name der Datei.

accessFlags$: Zugriffsflags, optional): Zeichenkette, bestehend aus einer Kombination der Buchstaben "rwx", die den Zugriff auf die Datei festlegen. Der Standard (accessFlags$ nicht angegeben) ist "rw": Datei zum Lesen und Schreiben öffnen.

alienFlags$: Fremdzugriffsflags, optional): Zeichenkette, bestehend aus einer Kombination der Buchstaben "rw", die den Zugriff auf die Datei festlegen. Der Standard (alienFlags$ nicht angegeben) ist "": Leerstring, keine Fremdzugriffe erlaubt.

Der Dateizeiger ist nach dem Öffnen einer Datei immer auf die Position 0 gesetzt. Wollen Sie Daten anhängen, müssen Sie ihn zunächst mit der Anweisung **FileSetPos** ans Dateiende setzen.

Die Systemvariable **fileError** wird gesetzt oder gelöscht. Beispiele und eine ausführliche Beschreibung der Flagzeichen finden Sie weiter unten.

Mögliche Fehlerbedingungen für **FileOpen** sind unter anderem:

- Die Datei existiert nicht.
- Die Datei existiert, aber sie ist von einem anderen Programm geöffnet, und dieses verweigert den Zugriff.

Flagzeichen für FileCreate und FileOpen

Die Befehle **FileCreate** und **FileOpen** erwarten ein oder zwei Zeichenketten, in denen durch einzelnen Buchstaben symbolisiert ist, was Sie wollen. Das Setzen einzelner Bits oder, wie hier, Zeichen, wird üblicherweise als das Setzen von "**Flags**" (Flaggen) bezeichnet.

accessFlags$ (Zugriffs-Flaggen-Zeichen) ist eine Zeichenkette, die bestimmt, wie die Datei angelegt oder geöffnet werden soll und ob auf die Datei lesend und/oder schreibend zugegriffen werden soll.

Mögliche **accessFlags** für **FileOpen**: **r**, **w**, **x** (x nur für Profis)
Mögliche **accessFlags** für **FileCreate**: **g**, **n**, **t**, **o**, **r**, **w**, **x** (x nur für Profis)

**alienFlags$** (Fremd-Zugriffs-Flaggen-Zeichen) ist eine Zeichenkette, welche die Zugriffsrechte für andere Programme bestimmt, während die Datei offen ist. Wird alienFlags$ nicht angegeben, so wird "" (leer, keine Zugriffsrechte) angenommen. Sie sollten davon nur abweichen, wenn es unbedingt erforderlich ist, da das System in diesem Fall viele Daten zwischenspeichern muss, was den Systemspeicher sehr belasten kann.

Mögliche **alienFlags$** für **FileOpen** und **FileCreate**: **r**, **w**

Zugriffsflags (accessFlags$) für FileCreate und FileOpen

- **r** "read": Aus der Datei kann gelesen werden.
- **w** "write" In die Datei kann geschrieben werden.

Wird weder '**r**' noch '**w**' angegeben, wird '**rw**' angenommen. Dies ist auch der Standard, wenn **accessFlags$** nicht angegeben wird.

Zugriffsflags (**accessFlags$**) nur für **FileCreate**

- **g** "GEOS": Es wird eine GEOS-Daten-Datei angelegt. Der Dateiname muss den GEOS-Namenskonventionen (max. 32 Zeichen) entsprechen. Wird 'g' nicht angegeben, wird eine DOS-Datei angelegt. Der Name muss dann der Konvention 8.3 entsprechen.
- **n** "neu": Nur Neuanlegen erlaubt. Es ist ein Fehler, wenn die Datei schon existiert, die Systemvariable fileError wird entsprechend gesetzt.
- **o** "open": Existiert die Datei schon, wird sie normal geöffnet. Die vorhandenen Daten bleiben erhalten.
- **t** "truncate" (= abschneiden): Existiert die Datei schon, wird sie abgeschnitten. Die vorhandenen Daten gehen verloren.

Wird weder 'n', 'o' noch 't' angegeben, wird 'n' angenommen. Dies ist auch der Standard, wenn "accessFlags$ nicht angegeben wird.

Beispiele:

```
FileOpen "info.txt", "r"  Nur Lesezugriff zugelassen
                Kein Fremdzugriff, da keine alienFlags$
                angegeben.
FileCreate "info.txt", "orw"  Öffnen einer vorhandenen Datei oder
                Anlegen einer Neuen mit Schreib- und Lesezugriff.
                Eventuell vorhandenen Daten bleiben erhalten.
                Kein Fremdzugriff, da keine alienFlags$
                angegeben.
FileCreate "info.txt", "gtw"  Öffnen einer vorhandenen GEOS-Daten-
                Datei oder Anlegen einer Neuen zum
                Schreibzugriff, ohne Lesezugriff. Eventuell
                vorhandene Daten werden gelöscht. Kein
                Fremdzugriff, da keine alienFlags$ angegeben.
```

Fremdzugriffsflags (**alienFlags$**) für FileCreate und FileOpen

- **r** "read": Andere Programme können aus der Datei lesen
- **w** "write" Andere Programme können in die Datei schreiben.

Wird weder '**r**' noch '**w**' angegeben, wird "" (keine Zugriffsrechte) angenommen. Dies ist auch der Standard, wenn **alienFlags$** nicht angegeben wird.

Tipps:

- Für die Flags sind Groß- und Kleinbuchstaben erlaubt.
- Die Reihenfolge der Flagzeichen ist egal.
- Stringausdrücke sind erlaubt.
- Überflüssige bzw. ungültige Zeichen werden ignoriert. Sie können den Flags-String so optisch strukturieren. Zu lange Zeichenketten (mehr als 32 Zeichen) können aber zu einem Laufzeitfehler führen.
- Verwenden Sie statt der Buchstaben keine Worte wie z.B. "write" statt "w". Sie würden im Beispiel die Datei zum Schreib **und** Lesezugriff öffnen, da **wr**ite die Buchstaben **w** und **r** enthält.

Sonstige **accessFlags**, nur für Profis:

- **x** "eXtended": Erweiterter Zugriff, z.B. auf den Header einer GEOS-Datei. **Achtung!** Ungültige Änderungen im Header können die Arbeit mit der Datei unmöglich machen! GEOS kann die Datei in bestimmten Fällen auch nicht mehr löschen. Sie sollten hier ganz genau wissen, was Sie tun. Der Programmierer von R-BASIC übernimmt **keinerlei Haftung**!

Beispiele für FileOpen und FileCreate:

Die folgenden Beispiele setzen voraus, dass eine FILE-Variable folgendermaßen definiert ist:

```
DIM  fh  AS  FILE
```

Anlegen einer DOS-Datei zum Lesen und Schreiben. Existiert die Datei schon, soll die Variable fileError gesetzt werden. Fremdprogramme sollen während dessen keinen Zugriff haben.

```
fh = FileCreate  "Info.TXT"
```

Diese Anweisung ist identisch mit: FileCreate "Info.TXT", "rw", ""

Anlegen einer GEOS-Daten-Datei zum Lesen und Schreiben. Es muss das Flag "g" angegeben werden, damit eine GEOS-Datei erzeugt wird. Existiert die Datei schon, soll die Variable fileError gesetzt werden. Fremdprogramme sollen während dessen Lesezugriff haben. Der Parameter "accessFlags$" muss angegeben werden um "alienFlags$" angeben zu können.

```
fh = FileCreate  "Meine Daten" ,  "g", "r"
```

Anlegen einer GEOS-Daten-Datei nur zum Schreiben. Existiert die Datei schon, wird sie geöffnet und der Inhalt verworfen. Fremdprogramme sollen während dessen keinen Zugriff haben.

```
fh = FileCreate  "Mein Daten-Logbuch" ,  "gtw"
```

Anlegen einer DOS-Datei zum Lesen und Schreiben. Existiert die Datei schon, wird sie geöffnet, der Inhalt bleibt erhalten. Fremdprogramme sollen während dessen keinen Zugriff haben.

```
fh = FileCreate  "Data.dat" ,  "o"
```

Öffnen einer Datei zum Lesen und Schreiben. Fremdprogramme sollen während dessen keinen Zugriff haben.

```
fh = FileOpen  "Info.TXT"
```

Diese Anweisung ist identisch mit: FileOpen "Info.TXT", "rw", ""

Öffnen einer GEOS-Daten-Datei zum Lesen und Schreiben. Beachten Sie, dass das Flag 'g' nicht angeben wird, R-BASIC erkennt selbstständig, dass es sich um eine GEOS-Datei handelt. Fremdprogramme sollen während dessen Lesezugriff haben. Der Parameter "accessFlags$" muss angegeben werden um "alienFlags$" angeben zu können.

```
fh = FileOpen  "Meine Daten" ,  "rw", "r"
```

Öffnen einer Datei nur zum Lesen. Fremdprogramme sollen während dessen keinen Zugriff haben.

```
fh = FileOpen  "Daten.TXT" ,  "r"
```

Tipp 1:
Wenn Sie nicht sicher sein können, ob die Datei schon existiert, verwenden Sie FileCreate mit den accessFlags **o** (open) oder **t** (truncate). Um eventuell vorhandene Daten nicht zu verlieren, verwenden Sie das Flag **o**.

```
fh = FileCreate  "Data.dat" ,  "o"
```

Tipp 2:
Nach dem Öffnen oder Anlegen einer Datei sollten Sie prüfen, ob die Operation erfolgreich war. Das können Sie mithilfe der Funktion **NullFile**() (siehe unten) oder der globalen Variablen **fileError** tun.

```
fh = FileCreate  "Data.dat" ,  "o"
IF fh = NullFile() THEN
    MsgBox "Fehler beim Anlegen der Datei. FehlerCode " + \
    ErrorText$(fileError)
END IF
```

```
fh = FileOpen  "Info.TXT"
IF fileError <> 0 THEN
    MsgBox "Fehler beim Öffnen der Datei. FehlerCode " + \
    ErrorText$(fileError)
END IF
```

FileClose

Schließt eine offene Datei. Alle evt. noch im Hauptspeicher befindlichen Daten werden auf den Datenträger geschrieben. Nach dem Schließen haben andere Programme wieder den vollen Zugriff auf die Datei, die von **FileOpen** bzw. **FileCreate** gesetzten Restriktionen sind aufgehoben.

Syntax: **FileClose** **<fh>**
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiel:

```
DIM  fh  AS FILE
fh = FileOpen "info.txt"
...
FileClose fh
```

NullFile

Liefert eine "leere" Dateivariable zurück, dient also zum Löschen einer Dateivariablen oder zum Prüfen, ob sie leer ist.

Syntax: **<dateiVariable>** = **NullFile**()
Die Klammern sind erforderlich.
<dateiVariable>: Variable vom Typ FILE.

Nachdem die Datei geschlossen wurde (FileClose), sollten Sie der Dateivariablen mit Hilfe der Funktion NullFile() die Information "keine Datei" zuweisen.

```
DIM  fh AS FILE
....
FileClose fh
fh = NullFile ()
```

Sie können dann prüfen, ob eine Datei noch offen ist:

```
IF  fh <> NullFile() THEN ...
```

Verwenden Sie im Falle eines Programmierfehlers eine schon geschlossene Datei nochmals, z.B. in der Reihenfolge

```
FileClose fh
x = FileRead ( fh )
```

gibt es einen BASIC Laufzeitfehler und das Programm wird beendet.

9.4 Lesen und Schreiben von Binärdateien

FileRead, **FileWrite** und **FileInsert** sind Universalroutinen. Sie arbeiten mit allen Dateien und allen Typen von Variablen zusammen, einschließlich Strings und Strukturen. Dateien, die nicht aus einer Abfolge von Textzeilen bestehen werde allgemein als binäre Dateien oder Binärdateien bezeichnet. Für die Arbeit mit Textdateien (lesen und schreiben von Textzeilen) gibt es zusätzlich darauf spezialisierte Routinen, siehe Kapitel 9.5.

FileRead

Liest eine bestimmte Anzahl von Bytes aus einer Datei. Der Dateizeiger wird hinter die gelesenen Daten gesetzt. Die Daten werden entsprechend dem Typ des Ausdrucks, in dem **FileRead** vorkommt, interpretiert.

Syntax: **<var>** = **FileRead** (**<fh>** , **size** [, **signed**])

<var>: Variable von beliebigem Typ. Die gelesenen Daten werden in diese Variable kopiert.
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.
size: Anzahl der zu lesenden Bytes. Bei numerischen Daten bestimmt size den Datentyp (1 = Byte, 2 = Word oder Integer, 4 = DWord, LONGINT oder WWFixed, 10 = Real).
Zulässige (Grenz~) Werte für size: 1 <= size <= 16384 (16 kByte)
signed: (optional): TRUE oder FALSE (Default: FALSE).
Nur für numerische Daten der Größe 2 Byte oder 4 Byte:
signed = TRUE: Daten sind Integer oder Longint (je nach size)
signed = FALSE: Daten sind Word oder DWord (je nach size)
Ist <var> vom Typ WWFixed wird signed ignoriert.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiele:

```
y = FileRead (fh, 2 )          ' Lesen eines Word-Wertes (2 Byte)
```

```
y = FileRead (fh, 2, TRUE )    ' Lesen eines Integerwertes (2 Byte)
```

```
text$  = FileRead (fh, 100)    ' 100 Byte lesen und als
                               ' Text ansehen.
  ! Ein String endet, wenn eine binäre Null (ASCII-Code Null)
  ! gelesen wird.
  ! LEN(stringVariable) kann daher kleiner als  'size' sein.
  ! Trotzdem werden  'size' Bytes gelesen.
```

```
DIM  time AS DateAndTime
time = FileRead (fh, SizeOf( time ) )      ' Lesen einer Struktur
```

FileWrite

Schreibt eine bestimmte Anzahl von Bytes in eine Datei, indem vorhandene Daten überschrieben werden. Bei Bedarf werden die Daten angehängt, d.h. die Datei wird verlängert. Der Dateizeiger wird hinter die geschriebenen Daten gesetzt.

Syntax: **FileWrite <fh> , <expression> , size [, signed])**

<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.
<expression>: Ausdruck von beliebigem Typ. Das Ergebnis des Ausdrucks (z.B. eine Zahl, eine Struktur oder ein Text) wird in die Datei kopiert.
size: Anzahl der zu schreibenden Bytes. Bei numerischen Daten bestimmt size den Datentyp (Byte, Word, DWord, Integer, LONGINT, Real), in den die Zahl vor dem Schreiben konvertiert wird.
Zulässige (Grenz~) Werte für size: 1 <= size <= 16384 (16 kByte)
signed: (optional): TRUE oder FALSE (Default: FALSE).
Nur für numerische Daten der Größe 2 Byte oder 4 Byte:
signed = TRUE: Daten sind Integer oder Longint (je nach size)
signed = FALSE: Daten sind Word oder DWord (je nach size)
Ist <var> vom Typ WWFixed wird signed ignoriert.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiele:

```
FileWrite (fh, n, 2 )              ' Schreiben eines Word-Wertes
```

```
FileWrite (fh, 110, 2, TRUE )  ' Schreiben eines Integer-Wertes
```

```
FileWrite (fh, text$ , 100)        ' 100 Byte als Text schreiben.
        !  ist text$ länger, wird der String abgeschnitten
        !  es wird keine Textendekennung (Null) geschrieben
        !  ist text$ kürzer, wird mit Nullen aufgefüllt
```

```
DIM  time AS DateAndTime
FileWrite (fh, time, SizeOf( time ) ) ' Schreiben einer Struktur
```

FileInsert

Fügt eine bestimmte Anzahl von Bytes in eine Datei ein, ohne das vorhandene Daten überschrieben werden. Die Datei wird dadurch automatisch verlängert. Der Dateizeiger wird hinter die geschriebenen Daten gesetzt.

Syntax: **FileInsert <fh> , <expression> , size [, signed])**
<fh> , <expression> , size [, signed]) siehe FileWrite.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.
Beispiele: siehe **FileWrite**.

Spezielle Hinweise zum Speichern von File-, Handle- und Objekt-Variablen in Dateien

FileRead, **FileWrite** und **FileInsert** können mit File-, Handle und Objektvariablen bzw. Ausdrücken umgehen. Verwenden Sie als Datengröße die Werte **SizeOf**(File) (=6), **SizeOf**(Handle) (=6), **SizeOf**(Object) (=8).

Es ist jedoch im Normalfall nicht sinnvoll, diese Werte in einer Datei zu speichern, da die enthaltenen Werte nur zeitlich begrenzt gültig sind.

- **File-Variablen** sind nur solange gültig, wie die Datei geöffnet ist. Wird die Datei nach dem Schließen erneut geöffnet, ist der Inhalt der Dateivariablen **NICHT** mit dem vom ersten Mal identisch.

- **Handle-Variablen** sind nur solange gültig, wie das von Ihnen bezeichnete Objekt bzw. die dahinter stehende Datenstruktur vorhanden ist. Beispielsweise gilt das von **FileFindFirst$** gelieferte Handle nur so lange, bis **FileFindDone** gerufen wird. Ein erneutes **FileFindFirst$** liefert eine anderes Handle, auch wenn es im gleichen Verzeichnis sucht.

  Die **Ausnahme** sind Handles, die von der VMFiles-Library geliefert werden und sich auf Datenstrukturen in einer VM-Datei beziehen. Diese müssen üblicherweise in der VM-Datei selber gespeichert werden und sind so lange gültig, wie die Datenstrukturen selbst in der VM-Datei sind.

- **Objektvariablen** sind so lange gültig, wie das Objekt, auf das die Variable verweist, existiert.

Spezielle Hinweise zum Speichern von numerischen Werten in Dateien

R-BASIC kennt 3 vorzeichenlose (BYTE, WORD und DWord) und 4 vorzeichenbehaftete (INTEGER, LONGINT, WWFixed und REAL) numerische Datentypen.

Um Zahlen mit diesen Datentypen sowohl in Dateien schreiben als auch aus Ihnen lesen zu können, gelten folgende Konventionen:

Lesen von numerischen Werten:

- Der Parameter **size** von **FileRead** bestimmt, wie viele Bytes gelesen und wie sie interpretiert werden (d.h. welchem Datentyp sie entsprechen). Zulässig sind die Werte
  1 Lesen eines Byte
  2 Lesen eines Word oder Integer
  4 Lesen eines DWord, LongInt oder WWFixed
  10 Lesen eines Real-Wertes
  Andere Werte können zu unerwarteten Ergebnissen führen. Die ersten beiden gelesenen Bytes werden als Word (bzw. Integer) interpretiert, die restlichen Bytes werden verworfen.

- Der Parameter **signed** bestimmt für die 2 und 4-Byte Datentypen, ob die Bytes als vorzeichenlose Zahl (**word** bzw. **DWord**, signed=FALSE, Default-Wert) oder als vorzeichenbehaftete Zahl interpretiert werden (**integer** bzw. **longint**, signed=TRUE, signed muss angegeben werden).
  Für WWFixed-Variablen wird signed ignoriert.
  Verwenden Sie am besten den gleichen Wert für **signed**, den Sie auch beim Schreiben verwendet haben.

- Nachdem die Daten gelesen und interpretiert wurden, werden sie von FileRead in eine Realzahl konvertiert, so dass FileRead innerhalb von beliebigen Ausdrücken wie jede andere Funktion verwendet werden kann.

Beispiele:

```
DIM  x, z AS Real
DIM  n AS Word

n = FileRead ( fh, 2 )            ' Lesen eines Word
x = FileRead ( fh, 2, TRUE )      ' Lesen eines Integerwertes,
                                  ' aber speichern als Real
x = FileRead ( fh, 4, TRUE )      ' Lesen eines LONGINT
x = FileRead ( fh, 4 )            ' Lesen eines DWord
z = 2.7 * FileRead ( fh, 2 ) + FileRead ( fh, 10 )
```

Schreiben von numerischen Werten:

Die Parameterkonventionen sind denen von **FileRead** analog.

- Der Parameter **size** von **FileWrite** bzw. **FileInsert** wie viele Bytes geschrieben werden und in welchen Datentyp die Zahl vorher konvertiert werden soll. Zulässig sind die Werte:
  1 Schreiben eines Byte
  2 Schreiben eines Word oder Integer
  4 Schreiben eines DWord oder LongInt
  10 Schreiben eines Realwertes
  Bei ungültigen Werten wird zunächst ein Word (bzw. Integer) geschrieben und der Rest mit Nullen aufgefüllt.

- Der Parameter **signed** bestimmt für die 2 und 4-Byte Datentypen, ob die Bytes als vorzeichenlose Zahl (**word** bzw. **dword**, signed=FALSE, Default-Wert) oder als vorzeichenbehaftete Zahl geschrieben werden (**integer** bzw. **longint**, signed=TRUE, signed muss angegeben werden).

  Dazu wird der von <expression> gelieferte (REAL~) Wert zunächst in den entsprechenden Datentyp konvertiert und dann in die Datei geschrieben.

- Bei einer Zahlenbereichsüberschreitung der 1, 2 und 4-Byte Datentypen (z.B. 100 000 für Integer oder 500 für Byte) werden intern die überschüssigen Bits ignoriert. Für die vorzeichenlosen Datentypen (**byte**, **word**, **dword**) entspricht das einer Modulo-Operation.
  Bei vorzeichenbehafteten Werten (**integer** und **longint**, dh. **signed** = TRUE) führt das dazu, dass aus einer zu großen positiven Zahl eine negative Zahl wird und umgekehrt.

Beispiele:

```
DIM  x AS Real
DIM  n AS Word

FileWrite ( fh, n, 2 )            ' Schreiben eines Word
FileWrite ( fh, x, 2, TRUE )  ' x  runden und Schreiben
                              ' als Integerwert
FileWrite ( fh, x, 4, TRUE )  ' Schreiben eines LONGINT, x vorher
                      runden
FileWrite ( fh, n, 4 )            ' Schreiben eines DWord, n
                      vorher
                              ' in DWord konvertieren
```

Tipp:
Wenn Sie mehr als eine einzige numerische Variable speichern wollen ist es sinnvoll, diese in eine Struktur zu packen. Damit ersparen Sie sich auch die manuelle Typunterscheidung, das R-BASIC dies bei Strukturen automatisch macht.

9.5 Lesen und Schreiben von Textdateien

R-BASIC verfügt über Spezialbefehle zum Lesen und Schreiben von Zeilen aus Textdateien. Diese Befehle arbeiten nur mit Stringvariablen zusammen und werden in diesem Abschnitt erklärt. Universelle Lese- und Schreibefehle finden Sie im Abschnitt 9.4.

FileReadLine$, FileWriteLine, **FileInsertLine** und **FileReplaceLine** sind auf Textzeilen, die durch ein Zeilenendezeichen abgeschlossen sind, spezialisiert. Das trifft für normale Textdateien zu. Zeilenendezeichen sind die ASCII-Codes 13 (Wagenrücklauf, Carriage return, CR) bzw. 10 (Zeilenvorschub, LineFeed, LF) oder eine Kombination davon, üblicherweise die Folge 13, 10 (CRLF).

Hinweis: Ein Texteditor und auch ein Text Objekt fügt häufig zur Gewährleistung der Lesbarkeit von Texten automatische (nur am Bildschirm vorhandene) Zeilenumbrüche ein. Die R-BASIC "Textzeilen" erscheinen daher als "Absätze" in einem Texteditor oder in einem Textobjekt.

FileReadLine$

Liest eine Textzeile aus einer Datei. Es werden maximal so viele Zeichen gelesen, wie die zu belegende Variable aufnehmen kann. Ist die Zeile länger (d.h. es wurde keine Zeilenendekennung gelesen), so wird fileError auf -11 (LINE_TO_LONG) gesetzt. Ihr Programm kann dann entsprechend reagieren. Die fehlenden Zeichen werden nicht übergangen, das nächste FileReadLine$ liest sie ein.

Syntax: **<z$>** = **FileReadLine$** (**<fh>** [, **mode**])
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.
mode: (optional): Behandlung der Zeilen-Ende-Zeichen. Siehe Tabelle.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Mode-Konstanten für FileReadLine$ (RLM = ReadLine Mode)

Konstante	Wert	Bedeutung
RLM_CLEAR	0	Defaultwert. Zeilen-Ende-Zeichen abschneiden.
RLM_REPLACE_TO_CR	1	Zeilen-Ende-Zeichen durch CR (ASCII-Code 13) ersetzen. Dieser Code wird von GEOS-Textobjekten als Zeilenendezeichen verwendet.
RLM_SET_CR	2	Zeilen-Ende-Zeichen entfernen und in jedem Fall ein CR-Zeichen (Code 13) anhängen, auch wenn kein Zeilenendezeichen vorhanden war (z.B. am Ende einer Datei).

RLM_DONT_CHANGE	3	Text nicht ändern, Zeilenendezeichen bleiben erhalten.

Hinweise:
- Als Zeilenendekennung werden CR (13), LF (10), CRLF und LFCR erkannt.
- Am Dateiende wird eine fehlende Zeilenendekennung akzeptiert und fileError auf Null (Kein Fehler) gesetzt.
- Lesen über das das Dateiende hinaus setzt fileError auf 128 (SHORT_READ_WRITE), siehe Beispiel 2

Beispiele:

```
z$ = FileReadLine$ (fh)                    ' eine Zeile lesen

WHILE   fileError = 0
    z$ = FileReadLine$ (fh)            ' eine Zeile lesen
    IF fileError <> 0 THEN Print z$    ' und ausgeben
    WEND

z$ = FileReadLine$ (fh, RLM_DONT_CHANGE)
                                       ' eine Zeile incl.
                                       ' CR/LF lesen
```

FileWriteLine

Schreibt eine Textzeile, indem vorhandene Daten überschrieben werden. Bei Bedarf wird die Textzeile angehängt, d.h. die Datei wird verlängert.

Syntax: **FileWriteLine** **<fh>**, **zeile$** [, **mode**])
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.
zeile$: zu schreibender Text
mode: (optional): Behandlung der Zeilen-Ende-Zeichen. Siehe Tabelle.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Mode-Konstanten für FileWriteLine, FileInsertLine, FileReplaceLine

Konstante	Wert	Bedeutung
WLM_APPEND_CRLF	0	Defaultwert. Zeilenende-Zeichenfolge CRLF (13, 10) anhängen.
WLM_CR_TO_CRLF	1	CR-Codes (ASCII-Code 13) durch CRLF (Folge 13, 10) ersetzen. Dadurch werden Texte, die von GEOS-Textobjekten kommen, zur Verwendung in DOS-Dateien angepasst.
WLM_SET_TO_CRLF	2	CR-Codes durch CRLF ersetzen, wie mode WLM_CR_TO_CRLF. Unterschied: War am Textende kein CR-Code, so wird ein zusätzliches CRLF angehängt.

WLM_DONT_CHANGE	3	Text nicht ändern, Zeilenendezeichen bleiben erhalten.

WLM_APPEND_LF	4	Zeilenende-Zeichen LF (10) anhängen.
WLM_CR_TO_LF	5	CR-Codes (ASCII-Code 13) durch LF (ASCII-Code 10) ersetzen. Dadurch werden Texte, die von GEOS-Textobjekten kommen, zur Verwendung in Linux und macOS angepasst
WLM_SET_TO_LF	6	CR-Codes durch LF ersetzen, wie mode WLM_CR_TO_LF. Unterschied: War am Textende kein CR-Code, so wird ein zusätzliches LF angehängt.

Beispiele:

```
FileWriteLine  fh, "Hallo Welt"            '  CRLF automatisch
                                           ' anhängen
FileWriteLine  fh, "Hallo Welt\r", WLM_SET_TO_CRLF
                                           ' '\r' (CR) durch CRLF
                                           ' ersetzen
FileWriteLine  fh, "\r\r\r", WLM_SET_TO_CRLF
                                           ' 3 Leerzeilen
FileWriteLine  fh, \
        "Text geschrieben von R-BASIC", WLM_SET_TO_CRLF
FileWriteLine  fh, "Letzte Zeile", WLM_DONT_CHANGE
                                           ' kein CRLF anhängen
```

FileInsertLine

Schiebt eine Textzeile in die Datei ein. Die Datei wird dadurch verlängert.

Syntax: **FileInsertLine** **<fh>**, **zeile$** [, **mode**])
<fh>, zeile$ [, mode]): siehe **FileWriteLine**

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

FileReplaceLine

Ersetzt eine Textzeile in einer Datei durch eine andere. Die ersetzte Zeile darf maximal 1024 Zeichen lang sein (R-BASIC Begrenzung für Strings).

Syntax: **FileReplaceLine** **<fh>**, **zeile$** [, **mode**])
<fh>, zeile$ [, mode]): siehe **FileWriteLine**

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

9.6 Sonstige Funktionen

Positionieren des Dateizeigers

Während die Datei geöffnet ist, gibt es einen "**Dateizeiger**", der die Position bestimmt, an der Daten gelesen oder geschrieben werden. Der Zugriff auf den Dateizeiger einer Datei erfolgt über eine Variablen vom Typ FILE, die von **FileOpen** bzw. **FileCreate** geliefert wird.

FileGetPos

Liest die aktuelle Position des Dateizeigers aus. Der zurückgegebene Wert liegt im Bereich von 0 (Dateianfang) bis 4294967295, da Dateien maximal 4 GByte groß werden können.

Syntax: **<numVar>** = **FileGetPos** (**<fh>**)

<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiel:

```
' Herausfinden, ob man schon am Dateiende ist
IF  FileSize(fh) = FileGetPos(fh) THEN Print "Dateiende erreicht"
```

FileSetPos

Setzt den Dateizeiger an eine bestimmte Position. Die folgenden File~ Operationen lesen bzw. schreiben ab dieser neuen Position.

Syntax: **FileSetPos** **<fh>**, **position** [, **fromEnd**]

<fh>: Dateivariable, bestimmt die betroffene Datei.
position: neue Dateiposition
fromEnd: bestimmt den Positionierungsmodus

- nicht angegeben oder Null: Ab Dateianfang
- ungleich Null: Ab Dateiende
  Achtung! "position" muss hier **negativ** sein, damit Sie eine Position vor dem Dateiende anwählen. Ist "position" positiv, ist das Ergebnis unbestimmt. Manchmal wird die Datei verlängert, manchmal nicht.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiele:

```
Setzen des Dateizeigers an den Dateianfang
  FileSetPos  fh, 0

Setzen des Dateizeigers ans Dateiende, so dass FileWrite Daten an
  die Datei anhängen kann:
  FileSetPos  fh, 0, TRUE

Setzen des Dateizeigers 100 Byte vor das Dateiende. Der Parameter
  "position" ist hierzu negativ.
  FileSetPos  fh, -100, TRUE

Verschieben des Dateizeiger um 10 Byte nach hinten.
  FileSetPos  fh, FileGetPos(fh) + 10

Setzen des Dateizeigers hinter das 2. Byte (ab Dateianfang). Das
  erste Byte hat die Position 0, das zweite die Position 1
  FileSetPos fh, 2
```

Tipps & Tricks:

- Benutzen Sie FileSize, um die aktuelle Größe der Datei zu ermitteln.
- Setzen Sie den Dateizeiger hinter das Dateiende (auf einen Wert, der größer ist, als der von FileSize gelieferte), so ist das Ergebnis unbestimmt.
  Anmerkung: In einigen Fällen wird die Datei verlängert (und die neu angehängten Bytes mit Nullen initialisiert), in anderen Fällen wurde die Dateilänge nicht geändert.

Weitere Dateioperationen

FileResize

Einfügen oder Löschen von Bytes an der aktuellen Position einer Datei. Die dahinter folgenden Daten werden automatisch verschoben. Eingefügte Bytes werden mit Null initialisiert. Der Dateizeiger wird hinter die eingefügten Bytes gesetzt. **FileResize** eignet sich auch, um Null-Bytes an eine Datei anzufügen.

Syntax: **FileResize <fh>, bytesToDelete, bytesToInsert**

<fh>:	Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.
bytesToDelete:	Anzahl zu löschender Bytes. Maximal 2 GByte.
bytesToInsert:	Anzahl einzufügender Bytes. Maximal 2 GByte.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiele:

```
FileResize  fh,  200,  0  ' 200 Bytes löschen. Der Dateizeiger
                    bleibt an der aktuellen Position.

FileResize  fh,  0,  200  ' 200 Bytes einfügen, der Dateizeiger
                    steht hinter den eingefügten Bytes.

FileResize  fh,  100,  200    ' 100 Byte löschen und durch 200
                    Null-Bytes ersetzen. Die Datei wird um 100
                    Byte verlängert, der Dateizeiger wird hinter
                    die 200 Null-Bytes gesetzt.

FileResize    fh,  100,  40   ' 100 Byte löschen und durch 40
                    Null-Bytes ersetzen. Die Datei wird um 60
                    Byte kürzer, der Dateizeiger wird hinter die
                    40 Null-Bytes gesetzt.

FilePos  fh, 0, TRUE      ' Dateizeiger ans Dateiende
FileResize fh, 0, 800     ' 800 Null-Bytes anhängen
```

FileTruncate

FileTruncate schneidet die Datei an der Position des aktuellen Dateizeigers ab.

Syntax: **FileTruncate <fh>**
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

Beispiel:

```
! Einkürzen einer Datei auf die Länge Null, d.h. alle Daten
  löschen.
FileSetPos fh, 0
FileTruncate  fh
```

FileCommit

Bewirkt, dass alle in Daten der Datei unverzüglich auf die Platte geschrieben werden. Das GEOS-System hält aus Performance-Gründen viele Daten oft bis zum Schließen der Datei im Speicher, d.h. die Daten kommen manchmal erst beim **FileClose** wirklich auf der Platte an. **FileCommit** ist sinnvoll, wenn mehrere Programme gleichzeitig auf die gleiche Datei zugreifen und man kann damit einem Datenverlust im Falle eines Systemabsturzes vorbeugen.

Syntax: **FileCommit <fh>**
<fh>: Variable (oder Funktion) vom Typ FILE. Bezeichnet die Datei.

Die Systemvariable **fileError** wird gesetzt oder gelöscht.

10 Arbeit mit Laufwerken und Datenträgern

Mit **DiskWriteable**, **DiskSpace**, **DiskExist** und **DriveInfo** erhalten Sie Informationen über Datenträger oder Laufwerke. **DiskGetName$** und **DiskRename** arbeiten mit der Datenträgerbezeichung. Diese Befehle setzen alle die Systemvariable **fileError** (z.B. wenn das Laufwerk nicht existiert) oder löschen Sie, wenn kein Fehler auftrat.

DiskWriteable

Prüft, ob sich ein beschreibbarer Datenträger im Laufwerk befindet. DiskWriteable liefert "wahr" (TRUE, -1) wenn sich ein beschreibbarer Datenträger im Laufwerk befindet. Existiert das Laufwerk nicht, befindet sich kein, oder kein formatierter Datenträger im Laufwerk, liefert DiskWriteable "falsch" (FALSE, 0).

Syntax: **<numVar>** = **DiskWriteable** (**lw$**)

lw$: Laufwerksbezeichnung, z.B. "A:" oder "D:"
<numVar> numerische Variable

Beispiel:

```
IF  DiskWriteable("a:") = 0  THEN Print "Die Diskette ist
  schreibgeschützt."
```

DiskSpace

Prüft den auf einem Datenträger verfügbaren Platz.

Syntax: **<numVar>** = **DiskSpace** (**lw$** [, **all**])

lw$: Laufwerksbezeichnung, z.B. "A:" oder "D:"
all: (optional) Wenn angegeben und ungleich Null, liefert DiskSpace den insgesamt auf dem Datenträger vorhandenen Platz.
Wenn nicht angegeben oder gleich Null, liefert DiskSpace den freien Speicherplatz auf dem Datenträger.
<numVar> numerische Variable

Wenn mehr als 2 GB verfügbar bzw. vorhanden sind, liefert DiskSpace immer den Maximalwert von 2 147 418 112 Byte zurück.

Beispiel:

```
Print  "Speicherstatus von Laufwerk C:"
Print  DiskSpace ( "C:", TRUE) " Bytes insgesamt"
Print  DiskSpace ( "C:" ) " Bytes verfügbar"
```

DiskExist

Prüft, ob sich ein formatierter Datenträger im Laufwerk befindet. Wenn Sie wissen wollen, ob ein bestimmtes Laufwerk existiert, verwenden Sie **DriveInfo** (unten). DiskExist liefert "wahr" (TRUE, -1), wenn ein formatierter Datenträger im Laufwerk ist. Das gilt auch für Festplatten. Befindet sich kein oder nur ein unformatierter Datenträger im Laufwerk, liefert DiskExist "falsch" (FALSE, 0).

Syntax: **<numVar>** = **DiskExist** (**lw$**)

lw$: Laufwerksbezeichnung, z.B. "A:" oder "D:"

Hinweis: DiskExist setzt die Systemvariable **fileError** auf 0, wenn das Laufwerk existiert aber kein Datenträger enthalten ist.

Beispiel:

```
IF  DiskExist("a:")  = 0  THEN Print "Legen Sie eine formatierte
  Diskette ein!"
```

DriveInfo

Liefert ausführliche Informationen über das angegebene Laufwerk. DriveInfo gibt Null zurück, wenn das Laufwerk nicht existiert, andernfalls einen Wert ungleich Null.

Syntax: **<numVar>** = **DriveInfo** (**lw$**)

lw$: Laufwerksbezeichnung, z.B. "A:" oder "D:"

Beispiel:

```
IF  DriveInfo("H:") = 0 THEN Print "Laufwerk H: existiert nicht"
```

Existiert das Laufwerk, enthält der Rückgabewert vielfältige Informationen über das Laufwerk, wobei jedes einzelne Bit eine Bedeutung hat. Die Auswertung dieser Daten ist etwas für Experten. Kenntnisse im Umgang mit Bits und logischen Verknüpfungen sind hilfreich. R-BASIC unterstützt die Arbeit mit den wichtigsten Eigenschaften durch ein paar vordefinierte Konstanten.

Wert	R-BASIC Konstante	Bedeutung
15	DI_TYPE_MASK	Maske zum Herausfiltern des Laufwerkstyp
2	DI_FIXED	Laufwerkstyp: Festplatte
4	DI_CD_ROM	Laufwerkstyp: CD-ROM-Laufwerk
64	DI_REMOVABLE	Datenträger ist wechselbar (CD, Diskette)
2048	DI_READ_ONLY	"Nur-Lesen" - Datenträger

Tabelle: R-BASIC Konstanten für DriveInfo. Eine komplette Beschreibung finden Sie auf der nächsten Seite.

Verwenden Sie die Konstanten wie folgt:

Beispiel 1:

```
DIM     bitfeld AS word

bitfeld = DriveInfo ("D:")
IF  bitfeld = 0  THEN Print "Laufwerk D: existiert nicht"

! ## Abfragen für Experten
IF  bitfeld  AND   DI_REMOVABLE  THEN
  Print "Datenträger in D: ist entnehmbar"
  END  IF
IF  ( bitfeld  AND  DI_TYPE_MASK ) = DI_CD_ROM   THEN
  Print "Laufwerk D: ist ein CD-Laufwerk"
  END IF
```

Beispiel 2:

```
DIM info, type AS word

info = DriveInfo ("D:")

! ## Herausfinden des Laufwerkstypes
type = info AND DI_TYPE_MASK      ' Bits 4 .. 15 Null setzen
IF    type  AND DI_FIXED   THEN    Print "Festplatte"
IF    type  AND DI_CD_ROM  THEN    Print "CD-ROM Laufwerk"

! ## Entnehmbar und beschreibbar
IF    info  AND DI_REMOVABLE THEN Print "Entnehmbarer
                                                Datenträger"
IF    info  AND DI_READ_ONLY THEN Print "Nur-Lesen Datenträger"
```

Bedeutung der Bits im Rückgabewert von DriveInfo

DriveInfo liefert eine 16-Bit Wert, der ausführliche Informationen über das abgefragte Laufwerk enthält. Hier finden Sie eine komplette Liste der Bits. Kenntnisse im Umgang mit Bits und logischen Verknüpfungen sind hilfreich.
Hinweis: Die meisten Infos sind der PC/GEOS-SDK-Dokumentation ungeprüft entnommen. Für eventuelle Fehler in der PC/GEOS-SDK-Dokumentation kann R-BASIC nichts.

Bit-Nummer	R-BASIC Konstante	Bedeutung
0 - 3	DI_TYPE_MASK = 15	Laufwerkstyp 0 : 5,25" Diskette 1 : 3,5" Diskette 2 : Fixed (Festplatte) 3 : RAM-Laufwerk 4 : CD-ROM-Laufwerk 5 : 8" Diskette 16: unbekannter Typ
4		(unbenutzt / reserviert)
5		Netzwerklaufwerk
6	DI_REMOVABLE = 64	Datenträger ist wechselbar (CD, Diskette)
7		Laufwerk ist physisch vorhanden
8		"besetzt" - Laufwerk wird benutzt
9		"alias" - Laufwerk ist ein Alias für einen Pfad auf einem anderen Laufwerk
10		Datenträger ist formatierbar
11	DI_READ_ONLY = 2048	"Nur-Lesen" - Datenträger, z.B. CD
12		Laufwerk ist nicht über das Netzwerk sichtbar.
13		(unbenutzt / reserviert)
14		(unbenutzt / reserviert)
15		(unbenutzt / reserviert)

Tabelle: Bedeutung der Bits im Rückgabewert von **DriveInfo**.

DiskGetName$

Liest die Datenträgerbezeichnung.

Syntax: **<name$>** = DiskGetName$ (**lw$**)
Parameter: lw$: Laufwerksbezeichnung
z.B. "A:" oder "D:"

Fehlerbedingung: Die Systemvariable fileError wird gesetzt oder zurückgesetzt.

Beispiel:

```
DIM s$, n$
  s$ = "a:"
  n$ = DiskGetName$( s$ )
  IF fileError THEN
    Print "Fehler beim Lesen der Bezeichnung von ";s$
  ELSE
    Print "Datenträger im Laufwerk "; s$; "heißt ";n$
  END IF
```

DiskRename

Schreibt die Datenträgerbezeichnung.

Syntax: DiskRename **lw$**, **name$**
Parameter: lw$: Laufwerksbezeichnung, z.B. "A:" oder "D:"
name$: Neue Datenträgerbezeichnung

Fehlerbedingung: Die Systemvariable fileError wird gesetzt oder zurückgesetzt.

Beispiel:

```
DIM s$
  s$ = "a:"
  DiskRename s$, "Paul"
  IF fileError THEN
    Print "Fehler beim Setzen der Bezeichnung von ";s$
  ELSE
    Print "Datenträger im Laufwerk "; s$; "heißt jetzt";
  DiskGetName$(s$)
  END IF
```

(Leerseite)

11 Portzugriffe

Achtung! Die Befehle greifen direkt auf die Hardware des Computers zu!

INP

Die Funktion INP (= Input) greift auf die Hardware des Computers zu und liest ein Byte von einem I/O-Port. Sie müssen sich mit der Hardware des Computers und den Portadressen sowie deren Bedeutung auskennen, um diesen Befehl nutzen zu können.

Syntax: **<numVar>** = INP (**port**)
Parameter: port: Port-Adresse, von der gelesen werden soll

OUT

Der Befehl OUT (= Output) greift auf die Hardware des Computers zu und schreibt ein Byte in einem I/O-Port. Sie müssen sich mit der Hardware des Computers und den Portadressen sowie deren Bedeutung auskennen, um diesen Befehl nutzen zu können.

Syntax: OUT **port**, **wert**
Parameter: port: Port-Adresse, auf die ausgegeben werden soll
wert: auszugebender Wert

Achtung: Eine Ausgabe ungültiger Werte auf bestimmte I/O-Ports des Computers könnte die Funktion des Computers schwer stören oder unmöglich machen. Der Programmierer von R-BASIC übernimmt keinerlei Haftung für Schäden, die auf eine fehlerhafte Verwendung der Befehle INP und OUT zurückgehen!

WAIT

Die Funktion WAIT (= Warte) greift auf die Hardware des Computers zu und wartet bis ein bestimmtes Bitmuster an einem I/O-Port anliegt. Sie müssen sich mit der Hardware des Computers und den Portadressen sowie deren Bedeutung auskennen, um diesen Befehl nutzen zu können.

Syntax: WAIT **port**, **mask** [, **xBits** [, **mode**]]
Parameter: port: Port-Adresse, von der gelesen werden soll
mask: Maske, welche Bits abgefragt werden sollen
xBits: Welche Bits davon Null sein müssen.
xBits ist optional. Vorgabewert ist Null.
mode: mode = 0: WAIT wartet **BIS** das gesuchte Bitmuster erscheint (Vorgabewert, wenn mode nicht angegeben).
mode = 1: WAIT wartet **SOLANGE** das gesuchte Bitmuster am Port anliegt.
Soll mode angegeben werden, so ist auch xBits anzugeben.

Funktion: WAIT wartet, bis die bitweise logische Verknüpfung des Bitmusters am abgefragten Port mit mask und (falls angegeben) xBits die geforderte Bedingung (mode) erfüllt.
"mask" bestimmt, welche Bits berücksichtigt werden sollen. Bits, die in mask nicht gesetzt sind, können beliebige Werte haben.
"xBits" bestimmt, welche Bits Null sein müssen. Bits, die in xBits nicht gesetzt sind, müssen Eins sein, damit die Bedingung erfüllt ist.

Intern werden "mask" und "xBits" (wenn angegeben) logisch XOR verknüpft. Der von "port" gelesene Wert mit "mask" logisch UND verknüpft. Die genauen Formeln lauten:

erwartung = (mask XOR xBits) AND mask
gelesen = INP(port) AND mask

mode = 0 wartet, bis das gelesene Bitmuster erscheint:
Warte bis erwartung = gelesen

mode = 1 wartet, solange das gelesene Bitmuster korrekt ist:
Warte solange wie erwartung = gelesen

Beispiele:
Wir nutzen: 1 ist binär 0001, 2 ist binär 0010, 3 ist binär 0011
x ist ein Bit, das gesetzt sein kann oder nicht (d.h. egal ob 0 oder 1)

Warten auf ein bestimmtes Bitmuster an Port p:

```
wait  p, 2     ' warten auf xx1x an Port p
wait  p, 2, 2  ' warten auf xx0x
wait  p, 3, 1  ' warten auf xx10
```

Warten solange ein bestimmtes Bitmuster an Port p anliegt:

```
wait  p, 2, 0, 1   ' warten solange xx1x an Port p
wait  p, 2, 2, 1   ' warten solange xx0x anliegt
wait  p, 3, 3, 1   ' warten solange xx00 anliegt
  ' WAIT setzt fort, wenn xx01, xx10 oder xx11 erscheint.
```

12 Focus und Target

Die Arbeit mit Focus und Target ist etwas für erfahrene Programmierer und nur in wenigen Fällen notwendig. Eine Ausnahme bildet die Implementation von speziellen Menüs wie dem "Bearbeiten" Menü. Diesem Thema ist deswegen ein eigenes Kapitel (Kapitel 13) gewidmet.

12.1 Überblick

Das GEOS-System benötigt einen oder mehrere Wege, auf dem Usereingaben zu dem Objekt geleitet werden, für das sie bestimmt sind. Neben dem Weg, den Mausereignisse gehen, und der in einem eigenen Kapitel beschrieben wird, sind das unter GEOS die Focus-Hierarchie und die Target-Hierarchie. Diese beiden Hierarchien arbeiten sehr ähnlich und werden im Folgenden beschrieben.

Focus
Die Focus-Hierarchie beschreibt, an welches Objekt Tastatureingaben des Nutzers gehen sollen. Das kann z.B. ein Textobjekt oder ein geöffnetes Menü sein. Man sagt, das Objekt, an das letztlich die Tastatureingaben gehen, "hat den Focus".
Eine Arbeit mit der Focus-Hierarchie ist nur sehr selten nötig, da Textobjekte automatisch damit umgehen können. Ein Beispiel wäre herauszufinden, welches VisObj-Objekt als letztes angeklickt wurde, also das "aktive" Objekt ist. Dieses hat nämlich den Focus (und auch das Target).

Target
Die Target-Hierarchie (engl. target = Ziel) beschreibt das Zielobjekt einer Operation, die der Nutzer ausführt. Ein gutes Beispiel ist ein Textobjekt, in das der Nutzer gerade etwas eingibt. Wenn der Nutzer ein Menü benutzt um die Größe des Texts im aktiven Textobjekt zu ändern, dann ist das Textobjekt das Ziel dieser "Operation". Das Gleiche gilt, wenn er einen Eintrag aus dem Edit-Menü wählt, um Text in die Zwischenablage zu kopieren oder von dort einzufügen. Man sagt, das Objekt, mit dem gearbeitet werden kann "ist das Target", manchmal auch "hat das Target".

Das letzte Beispiel zeigt auch sehr deutlich, warum zwei Hierarchien gebraucht werden. Klickt der User mit der Maus auf ein Menü verliert der Text den Focus, aber nicht das Target. Deswegen kann der Actionhandler des Menüeintrags entscheiden, mit welchem Objekt er interagieren soll - nämlich mit dem Target.

Sowohl die Focus- als auch die Target-Hierarchie sind an die Tree-Struktur der Objekte gekoppelt. Jedes Objekt kann genau eins seiner Children für den Focus- und eines für den Target-Pfad auswählen. Dasjenige Objekt, das am Ende dieses Pfades steht hat den Focus bzw. das Target. Das folgende Bild verdeutlicht das.

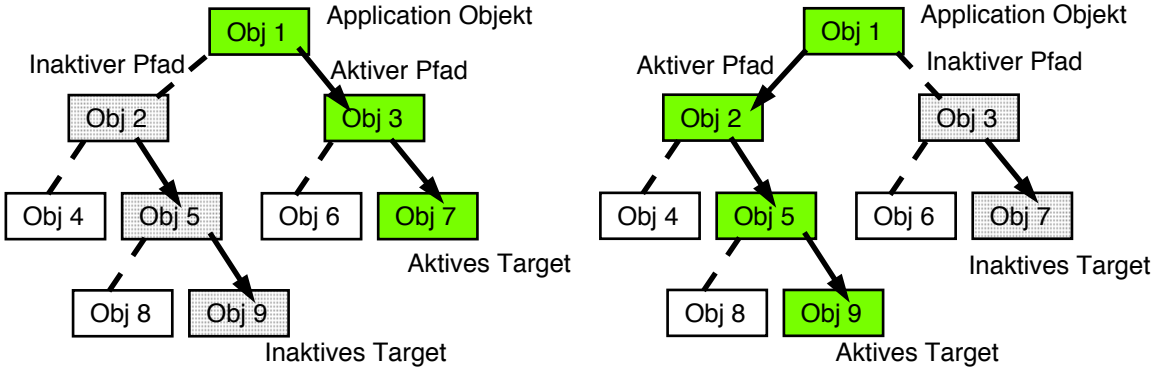


Bild 1: Aktive und inaktive Targets und Pfade

Die linke Abbildung zeigt den Ausgangszustand. Klickt der Nutzer jetzt mit der Maus auf Objekt 2 (z.B. eine Dialogbox) so wird Objekt 9 (z.B. ein Textobjekt in diesem Dialog) zum aktiven Target.

Jedes Mal, wenn ein Objekt Teil des aktiven Pfades wird oder den aktiven Pfad verlässt kann es eine Message aussenden (Handler **OnFocusChanged** oder **OnTargetChanged**). Auf diese Weise kann ein Programm stets über das aktuelle Target- oder Focus-Objekt informiert sein und bei Bedarf die UI anpassen, z.B. ein Edit-Menü enablen oder disablen.

Zur Arbeit mit Focus und Target stehen die folgenden Instancevariablen, Handlertypen und Systemvariablen zur Verfügung:

Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
OnFocusChanged	OnFocusChanged = **<Handler>**	nur schreiben
defaultFocus	defaultFocus	—
OnTargetChanged	OnTargetChanged = **<Handler>**	nur schreiben
targetable	targetable = TRUE | FALSE	lesen, schreiben
defaultTarget	defaultTarget	—

Action-Handler-Typen:

Handler-Typ	Parameter
FocusAction	(sender as object, hasFocus as integer)
TargetAction	(sender as object, hasTarget as integer)

Systemvariablen:

Variable	Inhalt
Focus	enthält das aktuelle Focus-Objekt
Target	enthält das aktuelle Target-Objekt

12.2 Arbeit mit dem Focus

Häufig wird der Hint **defaultFocus** verwendet um sicherzustellen, dass ein bestimmtes Objekt am Anfang den Focus hat. Eine weitergehende Arbeit mit dem Focus ist nur selten nötig. Alle Objekte können Standardsituationen automatisch handeln.

defaultFocus

Der Hint **defaultFocus** bewirkt, dass das Objekt am Anfang den Focus bekommt. Meistens reicht es, dem Objekt selbst diesen Hint zu geben, bei komplexen Baumstrukturen muss man ihn manchmal auch den Parents geben.

Syntax UI-Code:	**defaultFocus**

DefaultFocus ist auf GenericClass Level definiert und damit für alle Abkömmlinge der GenericClass verfügbar.

OnFocusChanged

Die Instancevariable **OnFocusChanged** enthält den Namen des Actionhandlers, der aufgerufen wird, wenn das Objekt den Focus erhält oder verliert. Er wird auch gerufen, wenn das Objekt Teil des aktiven Focuspfades wird oder vom aktiven zum inaktiven Pfad wechselt.
OnFocusChanged Handler müssen als **FocusAction** deklariert sein.

Der OnFocusChanged Handler ist für folgende Objektklassen definiert:

- Application
- Primary
- Menu
- Dialog
- Memo, InputLine, VisText, LargeText
- View
- BitmapContent
- Display, DisplayGroup
- VisContent
- VisObj

Der dem Handler übergebene Parameter "hasFocus" ist TRUE, wenn das Objekt den Focus erhalten hat (bzw. Teil des aktiven Pfades geworden ist), ansonsten ist er FALSE.

Beispiel: Wir wollen sicherstellen, dass ein InputLine Objekt disabled wird, wenn der Nutzer es verlassen hat.

UI-Code:

```
InputLine Text1
  defaultFocus
  OnFocusChanged = HandleFocus
End Object
```

BASIC-Code

```
FocusAction HandleFocus
  IF hasFocus = FALSE THEN
    Text1.enabled = FALSE
  End IF
End Action
```

Focus

Die globale Systemvariable **Focus** enthält das Objekt, das aktuell den Focus hat, d.h. das sich am Ende des aktiven Focuspfades befindet. Der Wert kann gelesen und geschrieben werden.

Beispiele

```
  IF Focus = MyObj THEN ...
  Focus.Capion$ = "Neuer Text"
```

```
' Ein bestimmtes Textobjekt als Focus auswählen, damit der
' User genau in dieses Objekt etwas einträgt
Focus = MyTextObject
```

Im Beispiel "Objekte\Visual Class\VisObj Keyboard Demo" finden Sie im OnDraw-Handler der VisObj-Objekte den folgenden Code. Damit wird (nur) um das aktive Objekt immer ein Rahmen gezeichnet.

```
  IF sender = Focus THEN
    Rectangle 5,5,MaxX-5, MaxY-5
  End IF
```

Achtung! Wenn Sie der Variablen Focus ein Objekt zuweisen, dass nicht am Ende eines Objektpfades steht (z.B. Focus = MyGroupWithChildren) kann das zu "komischem" Verhalten oder Systeminstabilität führen.

Hinweis: Es gibt Situationen, in denen R-BASIC das Focus-Objekt nicht identifizieren kann. Insbesondere ist das der Fall, wenn ein Number-Objekt oder ein Eintrag in einer DynamicList den "Focus" hat. Die Systemvariable Focus enthält dann einen Verweis auf ein "leeres" oder ein "internes" Objekt. Sie können die Situation prüfen, indem dem Sie die Instancevariable Class$ des Focus-Objekts abfragen. Focus.Class$ liefert in diesem Fall einen leeren String.

View: focusable

Für View-Objekte ist die numerische Instancevariable **focusable** definiert, die TRUE oder FALSE sein kann. Sie muss TRUE sein, damit das View (und damit sein Content) den Focus bekommen kann. Per Default ist sie TRUE.

12.3 Arbeit mit dem Target

Die Auswertung des Targets wird häufig benutzt um spezielle Menüs zu implementieren, z.B. das "Bearbeiten" Menü. Ein ausführlich kommentiertes Beispiel finden Sie im nächsten Kapitel. Eine weitergehende Arbeit mit dem Target ist etwas für erfahrende Programmierer und nur selten nötig. Alle Objekte können Standardsituationen automatisch handeln.

OnTargetChanged

Die Instancevariable **OnTargetChanged** enthält den Namen des Actionhandlers, der aufgerufen wird, wenn das Objekt zum Target wird oder diesen Status verliert. Er wird auch gerufen, wenn das Objekt Teil des aktiven Targetpfades wird oder vom aktiven zum inaktiven Pfad wechselt.
OnTargetChanged Handler müssen als **TargetAction** deklariert sein.

Der OnTargetChanged Handler ist für folgende Objektklassen definiert:

- Application
- Primary
- Memo, InputLine, VisText, LargeText
- Dialog
- View
- BitmapContent
- Display, DisplayGroup
- VisContent
- VisObj

Der dem Handler übergebene Parameter "hasTarget" ist TRUE, wenn das Objekt zum Target geworden ist (bzw. Teil des aktiven Pfades geworden ist), ansonsten ist er FALSE.

Beispiel:

Wenn Sie mehr als ein Textobjekt haben wird dieser Handler oft benutzt um die UI entsprechend den Attributen (Font, Textgröße, Farben usw.) anzupassen, die im Textobjekt dargestellt werden, mit dem der Nutzer gerade interagiert, d.h. dass das Target ist. Der Code zeigt, wie man im OnTargetChanged Handler ein Number-Objekt anspricht, so dass es die Größe des verwendeten Fonts anzeigt.

UI-Code:

```
Memo    Text1
  fontSize = 14
  fontID = FID_MONO
  defaultFocus
  OnTargetChanged = HandleTarget
End Object

Memo    Text2
  fontSize = 24
  fontID = FID_SANS
  OnTargetChanged = HandleTarget
End Object

Number PointInfoNumber
  Caption$ = "Aktuelle Font Größe:"
End Object
```

BASIC-Code

```
TargetAction HandleTarget
  IF hasTarget THEN
    ' UI updaten
    PointInfoNumber.value = sender.fontSize
  End IF
End Action
```

Target

Die globale Systemvariable **Target** enthält das Objekt, das aktuell das Target ist, d.h. das sich am Ende des aktiven Targetpfades befindet. Der Wert kann gelesen und geschrieben werden.
Die Variable wird oft in Actionhandlern verwendet, die auf das aktuelle Target wirken sollen. Ein ausführlich kommentiertes Beispiel finden Sie im nächsten Kapitel.

Beispiel: Ein Menü enthält eine OptionGroup, deren Actionhandler (namens ChangeSize) die Zeichengröße bei einem Textobjekt ändern soll.

```
LISTACTION ChangeSize
  Target.fontsize = selection
END ACTION   ' ChangeSize
```

Wenn man nicht sicher sein kann, dass das aktuelle Target ein Textobjekt ist, muss man vorher die Klasse abfragen.

```
LISTACTION ChangeSize
  IF Target.Class$ <> "MEMO" THEN RETURN
  Target.fontSize = selection
END ACTION   ' ChangeSize
```

Achtung! Wenn Sie der Variablen Target ein Objekt zuweisen (z.B. Target = MyObj), dass nicht am Ende eines Objektpfades steht oder nicht targetable ist, so kann das zu "komischem" Vehalten oder Systeminstabilität führen.

targetable (selten verwendet)

Die Instancevariable **targetable** enthält die Information, ob ein Objekt zum Target werden kann oder nicht. Der Wert ist per Default bei allen Objektklassen, für die ein OnTargetChanged Handler definiert ist (siehe oben) auf TRUE gesetzt, für alle anderen Objektklassen ist er per Default FALSE. Im Normalfall besteht keine Notwendigkeit daran etwas zu ändern. Setzen Sie z.B. bei einem der oben genannten Objekte targetable = FALSE so kann weder dieses Objekt noch seine Children (bzw. bei einem View sein Content) zum Target werden.

Syntax	UI-Code:	**targetable**
	Lesen:	<numVar> = <obj>.**targetable**
	Schreiben:	<obj>.**targetable** = TRUE I FALSE

Targetable ist auf GenericClass Level definiert und damit für alle Abkömmlinge der GenericClass verfügbar.
Für View-Objekte gilt: Um mit einem ViewControl zusammenzuarbeiten muss das View targetable sein. Außerdem muss das Bit VA_CONTROLLED in der Instancevariablen viewAttrs gesetzt sein.

defaultTarget

Der Hint **defaultTarget** bewirkt, dass das Objekt am Anfang das Target ist.

Syntax	UI-Code:	**defaultTarget**

DefaultTarget ist auf GenericClass Level definiert und damit für alle Abkömmlinge der GenericClass verfügbar.
Tipp:
- In vielen Fällen (z.B. Textobjekte) ist es **notwendig** statt defaultTarget den Hint **defaultFocus** zu verwenden.
- Wenn Sie View-Objekte haben, die mit einem ViewControl zusammenarbeiten sollen müssen Sie bei **genau einem** View den Hint defaultTarget setzen.

(Leerseite)

13 Implementieren von Menüs: Bearbeiten, Textgröße und andere

Dieses Kapitel zeigt an einem Beispiel, wie man Menüs implementiert, die mit dem Target zusammenarbeiten sollen. Das sind

- Ein "Bearbeiten" Menü (Kopieren, Einfügen usw.)
- Ein Menü "Größe" (Ändern der Schriftgröße)
- Ein Menü "Font" (Ändern des Textfonts)

Dazu schreiben wir ein Programm, dass neben den genannten Menüs drei Textobjekte enthält, wobei zwei davon mit den Menüs zusammenarbeiten sollen, eins aber nicht.

Der komplette Sourcecode, inklusive der hier nicht explizit dargestellten Objekte, finden Sie bei den Beispielen unter "Objekte\Allgemeines\Edit Menü & mehr".

Die Grundidee ist folgende:

- Klickt der Nutzer mit der Maus in ein Textobjekt so wird es zum Target. Das Objekt, das vorher das Target war verliert dabei diesen Status. Wir schreiben also einen **OnTargetChanged** Handler, der die Menüs updated, so dass sie den Zustand des neuen Targetobjekts reflektieren.
- Ist ein Objekt das Target, das nicht mit den Menüs zusammenarbeiten soll, werden die Menüs komplett disabled.
- Um die Buttons "Ausschneiden", "Kopieren" und "Löschen" zu verwalten benötigen wir außerdem einen **OnSelectionChanged** Handler, der gerufen wird, wenn der Nutzer in einem Textobjekt etwas selektiert oder deselektiert.
- Um den Button "Einfügen" zu verwalten benötigen wir einen **OnClpChange** Handler für das Applicationobjekt.

Die Textobjekte

Die ersten beiden Texte sollen mit den Menüs interagieren und haben deswegen sowohl einen OnTargetChanged als auch einen OnSelectionChanged Handler. Beachten Sie, dass die Handler für beide Textobjekte die gleichen sind. FontSize und FontID sind jedoch unterschiedlich. Der dritte Text verfügt über keinerlei Handler.

```
Memo    Text1
  fontSize = 14
  fontID = FID_MONO
  text$ = "Ein Mops kam in die Küche"
  defaultFocus
  OnTargetChanged = HandleTarget
  OnSelectionChanged = HandleSelection
  fixedSize = 200, 150
End Object

Memo    Text2
  fontSize = 24
  fontID = FID_SANS
  text$ = "und stahl dem Koch ein Ei."
  OnTargetChanged = HandleTarget
  OnSelectionChanged = HandleSelection
  fixedSize = 200, 150
End Object

Memo Text3
  text$ = "Ich interagiere nicht mit den Menüs!"
  ExpandWidth
End Object
```

Die Menüs

Das Edit-Menü ("Bearbeiten") enthält die Buttons für die entsprechenden Funktionen. Jeder Button hat seinen eigenen Actionhandler und einen Keyboard Shortcut gesetzt. Keyboard Shortcuts sind im Kapitel 3.1.4 des Objekthandbuchs beschrieben.

```
Menu EditMenu
  Caption$ = "Bearbeiten" , 0
  Children = CutButton, CopyButton, PasteButton, DeleteButton
End OBJECT

Button CutButton
  Caption$ = "Ausschneiden", 0
  ActionHandler = DoCut ' ButtonAction
  kbdShortcut = KSM_CTRL+ KSM_PHYSICAL+ ASC("x")
End OBJECT
Button CopyButton
  Caption$ = "Kopieren" , 0
  ActionHandler = DoCopy ' ButtonAction
```

```
  kbdShortcut = KSM_CTRL+ KSM_PHYSICAL+ ASC("c")
End OBJECT

Button PasteButton
  Caption$ = "Einfügen" , 0
  ActionHandler = DoPaste ' ButtonAction
  kbdShortcut = KSM_CTRL+ KSM_PHYSICAL+ ASC("v")
End OBJECT

Button DeleteButton
  Caption$ = "Löschen"' , 0
  ActionHandler = DoDelete ' ButtonAction
  kbdShortcut = &hF9A ' Siehe KeyCodes Library
End OBJECT
```

Die Menüs für Zeichengröße und Font sind prinzipiell gleich aufgebaut. Jedes Menü enthält eine RadioButtonGroup mit drei RadioButton-Objekten. Das Besondere ist, das der Identifier der RadioButton-Objekte gleichzeitig die Eigenschaft repräsentiert, die er einstellen soll. Die RadioButton-Objekte aus dem FontSelector sind Font-ID's und die aus dem SizeSelektor sind Punktgrößen.

```
Menu FontMenu
  Caption$ = "Font" , 0
  Children = FontSelector
End OBJECT

RadioButtonGroup FontSelector
  Children = FontOption1, FontOption2, FontOption3
  ApplyHandler = ChangeFont ' ListAction
  selection = FID_MONO
  End OBJECT
RadioButton FontOption1
  Caption$ = "Mono" : identifier = FID_MONO
  End OBJECT
RadioButton FontOption2
  Caption$ = "Sans" : identifier = FID_SANS
  End OBJECT
RadioButton FontOption3
  Caption$ = "Symbol" : identifier = FID_SYMBOLPS
  End OBJECT
```

Das Menü "SizeMenu" ist prinzipiell genauso aufgebaut.

```
Menu SizeMenu
  Caption$ = "Größe" , 0
  Children = SizeSelector
End OBJECT

RadioButtonGroup SizeSelector
  Children = SizeOption1, SizeOption2, SizeOption3
  orientChildren = ORIENT_VERTICALLY ' ORIENT_HORIZONTALLY
  ApplyHandler = ChangeSize ' ListAction
  selection = 14
```

```
  End OBJECT
RadioButton SizeOption1
  Caption$ = "14 pt" : identifier = 14
  End OBJECT
RadioButton SizeOption2
  Caption$ = "18 pt" : identifier = 18
  End OBJECT
RadioButton SizeOption3
  Caption$ = "24 pt" : identifier = 24
  End OBJECT
```

Das Application Objekt

Das Applicationobjekt muss einen OnClpChange Handler haben, damit das Clipboard überwacht werden kann.

```
Application DemoApplication
    Children = DemoPrimary
    OnClpChange = MonitorClipboard
END Object
```

Überwachen des Clipboard

Der OnClpChange Handler wird jedes Mal gerufen, wenn irgendeine Applikation Änderungen am Clipboard vornimmt. Das schließt unser eigenes Programm mit ein. Wir müssen daher nur nachschauen, ob ein Text im Clipboard ist und den "Einfügen" Button enablen oder disablen. Dazu verwenden wir die BASIC Routine ClipboardTest. Das Format der Daten, die sich im Clipboard befinden, wird durch eine eindeutige Kombination aus ManufacturerID (manufID) und Format-Nummer (formatNo) gekennzeichnet. Für Texte gilt: manufID = 0 (GeoWorks), formatNo. = 0 (TEXT). Mehr dazu finden Sie im Kapitel 5 "Arbeit mit der Zwischenablage".

```
SYSTEMACTION MonitorClipboard
  IF ClipboardTest(0, 0) then
    PasteButton.enabled = TRUE
  else
    PasteButton.enabled = FALSE
  End IF

END ACTION   ' MonitorClipboard
```

Sollte gerade ein Objekt das Target sein, dass nicht mit den Menüs zusammenarbeiten soll oder kann, so stört uns das hier nicht, da wir an anderer Stelle sicherstellen, dass das Menu-Objekt dann nicht enabled ist. Der PasteButton ist dann niemals aktiv, auch wenn er enabled ist.

Die anderen Buttons des Bearbeiten-Menüs

Alle anderen Buttons des Bearbeiten-Menüs müssen enabled werden, wenn der Nutzer Text selektiert hat, andernfalls müssen sie disabled werden. Da diese Abfrage mehrfach gebraucht wird lagern wir sie in eine SUB aus. Der Parameter textObj bezeichnet das aktuelle Targetobjekt, die Instancevariable selectionLen enthält die Anzahl der selektierten Zeichen.

```
SUB UpdateEditMenu (textObj as OBJECT)

  IF textObj.selectionLen THEN    ' d.h. selectionLen <> 0
    CopyButton.enabled = TRUE
    CutButton.enabled = TRUE
    DeleteButton.enabled = TRUE
  ELSE
    CopyButton.enabled = FALSE
    CutButton.enabled = FALSE
    DeleteButton.enabled = FALSE
  END IF

END SUB 'UpdateEditMenu
```

Die Textobjekt- Handler

Jetzt kümmern wir uns darum, was passiert, wenn der Nutzer ein Textobjekt anklickt. Das entsprechende Objekt wird dann zum Target. Vorher - und das ist sehr wichtig für uns - verliert das Objekt, das bis dahin Target war, jedoch seine Target-Status. Der OnTargetChanged Handler wird also zweimal gerufen: **zuerst** von dem Objekt das bis dahin Target war (mit dem Parameter hasTarget = FALSE) und **danach** von dem Objekt das jetzt Target wird (mit dem Parameter hasTarget = TRUE).
Wird er Handler also wegen einem Targetverlust gerufen disablen wir einfach alle Menüs, andernfalls enablen wir sie und updaten die UI.

Das hat einen weiteren Vorteil: Klickt der User auf ein Objekt, dass nicht mit den Menüs zusammenarbeiten kann oder soll (in unserem Fall Text3), so wird der OnTargetChanged Handler nur einmal gerufen (mit hasTarget = FALSE) und die Menüs bleiben so lange disabled, bis der Nutzer wieder in eins der Objekte Text1 oder Text2 klickt.

```
TARGETACTION HandleTarget

  if hasTarget = FALSE THEN
    Editmenu.enabled = FALSE
    Fontmenu.enabled = FALSE
    Sizemenu.enabled = FALSE
    return
    end if

  EditMenu.enabled = TRUE
  UpdateEditMenu (sender)
```

```
  FontMenu.enabled = TRUE
  FontSelector.selection = sender.fontID

  SizeMenu.enabled = TRUE
  SizeSelector.selection = sender.fontSize

END ACTION   ' HandleTarget
```

Außerdem müssen wir noch das Edit-Menü informieren, falls der Nutzer die Textselektion ändert.

```
TEXTACTION HandleSelection
  UpdateEditMenu ( sender )
END ACTION
```

Handler des Bearbeiten-Menüs

Nun müssen wir die ActionHandler der Buttons aus dem Bearbeiten-Menü implementieren. Da wir nicht wissen können, ob das aktuelle Target das Objekt Text1 oder Text2 ist verwenden wir als Ziel die globale Variable Target. Wenn die Handler der Menüs aufgerufen werden kann dies nur eines der beiden genannten Objekte sein, da wir vorne sichergestellt haben das die Menüs nur aktiv sind, wenn eines dieser Objekte das Target ist.

Die Handler an sich sind relativ einfach. Wir verwenden die für alle GenericClass Objekte definierten Clipboard-Methoden ClpCopy und ClpPaste. Ausschneiden entspricht einem Kopieren in das Clipboard mit anschließendem Löschen. Für das Löschen verwenden wir die Textobjekt Methode DeleteSelection. Außerdem müssen wir noch das Edit-Menü updaten, wenn wir etwas gelöscht haben. Dazu greifen wir wieder auf die globale Variable Target zurück

```
BUTTONACTION DoCut
  Target.ClpCopy
  Target.DeleteSelection
  UpdateEditMenu (Target)
END ACTION

BUTTONACTION DoCopy
  Target.ClpCopy
END ACTION

BUTTONACTION DoPaste
  Target.ClpPaste
END ACTION

BUTTONACTION DoDelete
  Target.DeleteSelection
  UpdateEditMenu (Target)
END ACTION
```

Die anderen Menü-Handler

Die Menüs - und damit die RadioButtonGroups - sind nur aktiv, wenn eines der Objekte Text1 oder Text2 das Target ist. Da die Identifier der RadioButton-Objekte ihre Funktion (eine FontID oder eine Schriftgröße) widerspiegeln werden die Actionhandler der RadioButtonGroups sehr einfach.

```
LISTACTION ChangeFont
  Target.fontID = selection
END ACTION   ' ChangeFont

LISTACTION ChangeSize
  Target.fontsize = selection
END ACTION   ' ChangeSize
```

Abschließende Überlegungen

Besondere Aufmerksamkeit verdient die Frage, ob die Menüs am Programmstart wirklich immer die korrekte Situation widerspiegeln. In unserem Fall müssen die Menüs die Eigenschaften des Objekts Text1 widerspiegeln, weil dieses Objekt den Hint defaultFocus gesetzt hat. Hier ist manchmal etwas Handarbeit (setzen der richtigen Startwerte) angesagt.

Der eleganteste und sicherste Weg um eventuell verbleibende Probleme zu umgehen ist, einen OnStartup Handler für das Applicationobjekt zu schreiben. Dort können Sie die UI-Objekte Ihren Vorstellungen nach anpassen.

```
Application DemoApplication
    Children = DemoPrimary
    OnStartup = StartupCode
  < ... >
END Object
```

```
SYSTEMACTION StartupCode
  UpdateEditMenu ( Text1 )
  < ... >
END ACTION   ' StartupCode
```

Für den Anfänger ist es oft sehr schwer, die Zusammenhänge zu überblicken. In unserem Fall stellt sich die Situation für einen erfahrenen Programmierer so dar: Für den Zustand der Menüs sind genau zwei Routinen zuständig: der Actionhandler MonitorClipboard und die SUB UpdateEditMenu. MonitorClipboard wird am Programmstart automatisch gerufen. Da das Objekt Text1 den Hint defaultFocus gesetzt hat wird es am Programmstart auch gleichzeitig zum Target. Damit wird der Handler HandleTarget mit dem Parameter hasTarget = TRUE am Programmstart gerufen, was den Aufruf von UpdateEditMenu zur Folge hat. Damit sind die Menüs auf dem aktuellen Stand. Wie gesagt, der Einsteiger überblickt so etwas nicht.

Deswegen die folgenden Tipps:

- Prüfen Sie den Zustand der Menüpunkte am Programmstart bewusst nach.
- Versuchen Sie, falls möglich, absichtlich verschiedene Zustände herzustellen um die Schwachstellen zu finden. Starten Sie das Programm zum Beispiel bewusst einmal mit und einmal ohne, dass sich Text im Clipboard befindet.
- Disablen Sie die Menüs per Default (enabled = FALSE), wenn Sie nicht sicher wissen, dass sie aktiv sein dürfen. Fälschlicher Weise disablete Menüs fallen bei einer Sichtprüfung eher auf als fälschlicher Weise enablete Menüs.