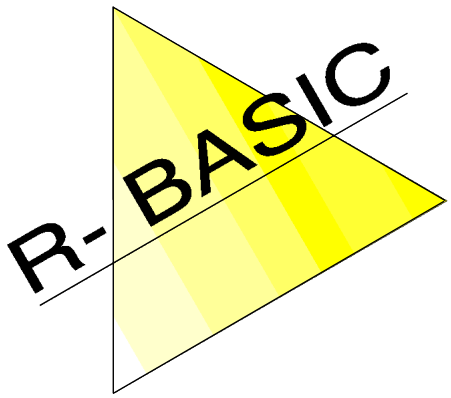
R-BASIC

Einfach unter PC/GEOS programmieren



Spezielle Themen

Volume 3
Tastatur, Document-Interface, Timer, Maus,
Abwärtskompatibiltät, Objekte individualisieren

Version 1.0

(Leerseite)

Inhaltsverzeichnis

(Leerseite)

14 Arbeit mit der Tastatur

Zur Entgegennahme von Nutzereingaben über die Tastatur stehen dem R-BASIC Programmierer drei Möglichkeiten zur Verfügung

1. Im **Klassischen BASIC** Mode wird die Tastatur direkt abgefragt. Dazu stehen z.B. die Befehle INPUT, Inkey$ und GetKey zur Verfügung. Unter einem objektorientierten System wie GEOS sollten Sie diesen Weg vermeiden. Insbesondere die ständige Abfrage der Tastatur in einer Schleife bremst das System massiv aus und erhöht die CPU-Last.
2. Der einfachste Weg ist die **Verwendung von Textobjekten** (Memo, InputLine, VisText oder LargeText). Diese Objekte können intelligent mit Tastatur und Maus umgehen und reichen für viele Zwecke völlig aus.
3. Der universellste Weg ist das Schreiben eines **Tastaturhandlers**. Tastaturhandler werden automatisch gerufen, wenn der Nutzer eine Taste drückt. Sie können die gerückte Taste mitlesen und entscheiden, ob Sie sie selbst behandeln oder einfach weitergeben wollen. Dieser Weg wird z.B. benutzt um ein Spielprogramm zu schreiben, dass mit der Tastatur gesteuert wird.

Dieses Kapitel widmet sich dem Schreiben von Tastaturhandlern, welche Möglichkeiten sich daraus ergeben und was es zu beachten gilt.

14.1 Überblick über Tastaturereignisse

Um mit den Tastaturhandlern arbeiten zu können sollte man mindestens in Grundzügen verstanden haben, wie GEOS mit Tastaturereignissen umgeht. Im Folgenden wir das Prinzip erklärt, für vollständige Informationen muss auf weiterführende Literatur verweisen werden.

Es gibt genau drei Situationen, in denen ein Tastaturereignis erzeugt wird:

1. Der Nutzer drückt eine Taste nieder. Dieses Ereignis heißt FIRST_PRESS.
2. Der Nutzer hält die Taste länger gedrückt. Dieses Ereignis heißt REPEAT_PRESS. Es wird immer wieder gesendet, solange der Nutzer die Taste unten hält. Die Häufigkeit (Tastatur-Wiederholungsrate) kann in den Voreinstellungen ausgewählt werden.
3. Der Nutzer lässt die Taste wieder los. Dieses Ereignis heißt RELEASE.

Bei jedem dieser Tastaturereignisse passiert intern folgendes:

1. Die Tastatur sendet die Nummer der Taste, den sogenannten **Scancode**, an den Computer. Dieser Scancode ist unabhängig vom eingestellten Tastaturlayout. Im Kapitel 14.3 finden Sie ein Bild mit den Scancodes einer PC-Tastatur.
2. Der Tastaturtreiber kennt das Tastaturlayout und wandelt den Scancode in den zugehörigen ASCII-Code (0 ... 255) um. Dabei werden die "Modifier"-Tasten wie Shift oder AltGr bereits berücksichtigt.

Tasten, denen kein ASCII-Code zugeordnet ist wie F1 oder die Cursortasten, erzeugen einen "erweiterten" ASCII-Code im Bereich von 65280 bis 65535 (hexadezimal &hFF00 bis &hFFFF).

3. Jetzt erzeugt der Tastaturtreiber ein "Tastaturereignis" für das System und gibt es als Message an das Application-Objekt des aktiven Programms weiter. Diese Message enthält unter anderem den ASCII-Code (das schließt erweiterte ASCII-Codes ein), den Typ des Ereignisses (FIRST_PRESS usw.) und den aktuellen Zustand der Steuertasten (z.B. linke Strg-Taste gedrückt).
4. Das Application-Objekt leitet das Ereignis an das Objekt weiter, dass den Focus hat.

Anmerkungen:

Zu Punkt 2:

Bei den erweiterten ASCII-Codes sind die höherwertigen 8 Bit immer gesetzt, die unteren 8 Bit enthalten die Information, z.B. den Steuercode der zur gedrückten Taste gehört. Man kann ihn abfragen indem man die oberen 8 Bit mit der AND-Operation ausblendet.

```
code = character AND 255            ' d.h. AND &hFF
```

Diese Codes sind oft, aber nicht immer identisch mit den für das PRINT-Kommando verwendeten Steuercodes (siehe Anhang, Kapitel A). Zum Beispiel ist der PRINT Code für "Cursor nach links" 14 (&h0E), der Tastencode ist jedoch 147 (&h93).

In der Library "KeyCodes" finden Sie symbolische Konstanten und die Werte für die erweiterten Tastencodes.

Zu Punkt 3:

Drückt der Nutzer die Tasten Shift + ’a’ wird der Buchstabe ’A’ erzeugt. Die Shift-Taste ist damit ausgewertet und wird NICHT mehr in das Tastaturereignis aufgenommen.

Es ist nun möglich sich unter R-BASIC an bestimmten Stellen in den Tastaturhandler einzuklinken und Tastaturereignisse zu "überschreiben". Das heißt man kann sie mitlesen, bei Bedarf verhindern dass sie weitergeleitet werden oder auch dem System eigene "Ereignisse" unterschieben. Der folgende Abschnitt beschreibt die Konzepte dahinter.

14.2 Schreiben eines Tastaturhandlers

Um sich in das Tastaturhandling einzuklinken muss man einen Tastaturhandler (auch Keyboardhandler) schreiben. Dazu stehen die folgenden Instancevariablen zur Verfügung:

Variable	Syntax im UI-Code	Im BASIC-Code
OnKeyPressed	OnKeyPressed = **<Handler>**	nur schreiben
inputFlags	inputFlags = **<numWert>**	lesen, schreiben

Handler-Typ	Parameter
KeyboardAction	(sender as object, character as word, keyState as word, keyFlags as byte, scanCode as byte)

Der OnKeyPressed Handler wird gerufen, wenn das Objekt ein Tastaturereignis erhält. Er muss als KeyboardAction deklariert werden. Die inputFlags bestimmen, in welchen Fällen das Ereignis vom Objekt selbst, vom BASIC Handler oder von beiden behandelt werden soll.

Für die folgenden Objektklassen sind Keyboardhandler und inputFlags definiert:

- Application
- Memo, InputLine, VisText und LargeText
- View
- VisContent
- BitmapContent
- VisObj

Wichtig: Es ist grundsätzlich so, dass zuerst das Objekt das Tastaturereignis behandelt (bzw. weiterleitet) und erst danach der BASIC Handler gerufen wird. Im Abschnitt 14.4 ist beschrieben, wie man trotzdem bestimmte Zeichen herausfiltern kann.

Einige der Objekte (z.B. alle Textobjekte) behandeln das Ereignis per Default selbst, andere (z.B. View oder VisContent) leiten es nur an untergeordnete Objekte weiter. Details oder Besonderheiten zum Tastaturhandling der einzelnen Objekte finden Sie in den entsprechenden Kapiteln zu den Objekten.

OnKeyPressed

Der OnKeyPressed Handler wird aufgerufen, wenn das Objekt ein Tastaturereignis erhält. Der Handler muss als KeyboardAction deklariert sein. Der BASIC Handler wird erst gerufen nachdem das Objekt das Ereignis selbst behandelt bzw. an untergeordnete Objekte weitergeleitet hat.

Syntax	UI- Code:	**OnKeyPressed** = **<Handler>**
	Schreiben:	**<obj>**.**OnKeyPressed** = **<Handler>**

KeyboardAction Handler haben die folgenden Parameter:

sender:	Das Objekt, welches den Handler aufgerufen hat
character:	ASCII-Code oder erweiterter ASCII-Code der Taste Tipp: In der Library "KeyCodes" finden Sie symbolische Konstanten für die erweiterten ASCII-Codes.
keyState:	Information, welche Status- oder Steuertasten aktuell gedrückt sind.
keyFlags:	Information ob der Nutzer die Taste gerade gedrückt hat, sie gedrückt hält oder gerade losgelassen hat.
scanCode:	Der Scancode der gedrückten Taste. Für Ereignisse, die mit der Methode KbdEvent erzeugt wurden hat scanCode den Wert Null.

KeyState enthält genau die Informationen, die man auch mit der BASIC Funktion GetKeyState erhalten kann. Der Zugriff auf den Parameter keyState ist jedoch wesentlich schneller. KeyState sind Bitflags. Jedes Bit hat eine eigene Bedeutung. Das niederwertige Byte enthält den "Shift"-Status, das höherwertige Byte (Bitwerte 256 und aufwärts) den "Toggle"- Status.
Folgende Werte bzw. Konstanten sind definiert:

Konstante (Shift-State)	Wert	(hex.)	Bedeutung
–	1	&h01	Feuertaste 1 am Joystick
–	2	&h02	Feuertaste 2 am Joystick
KS_RSHIFT	4	&h04	Rechte Shift-Taste
KS_LSHIFT	8	&h08	Linke Shift-Taste
KS_RCTRL	16	&h10	Rechte Strg-Taste
KS_LCTRL	32	&h20	Linke Strg-Taste
KS_RALT	64	&h40	Rechte Alt-Taste
KS_LALT	128	&h80	Linke Alt-Taste

Konstante (Toggle-State)	Wert	Bedeutung
KS_SCROLL_LOCK	256 (&h100)	Scroll-Lock-Taste (Rollen) eingerastet
KS_NUM_LOCK	512 (&h200)	Num-Lock-Taste eingerastet
KS_CAPS_LOCK	1024 (&h400)	Shift-Lock Taste eingerastet

Anmerkungen:

1. Die Bits werden vom Host-System an GEOS und von dort an den R-BASIC Handler übergeben. Nicht verwendete Bits sind intern verwendet und könnten gesetzt sein oder nicht.
2. Die Bits für "Modifier"-Tasten wie Shift oder AltGr sind i.A. nicht gesetzt, auch wenn die Tasten gedrückt sind. Sie wurden bei der Erzeugung des ASCII-Codes vom Tastaturtreiber "geschluckt".
3. Die Toggle-State Bits enthalten die Information, ob der entsprechende Zustand eingerastet ist oder nicht. Auf der Tastatur sollten dann die entsprechenden Leuchtdioden aktiv sein. Erfahrungsgemäß haben verschiedene Hostsysteme und/oder Emulatoren damit aber Probleme, so dass die LED's nicht immer den aktuellen Zustand widerspiegeln. Beispielsweise startet die DosBox unter Windows 7 immer mit dem Zustand "NumLock nicht aktiv", obwohl die LED leuchtet und man muss die Taste zweimal betätigen um den NumLock Zustand zu ändern.

KeyFlags sind Bitflags. Jedes Bit hat eine eigene Bedeutung. Folgende Werte bzw. Konstanten sind definiert.

Konstante	Wert	Bedeutung
KF_STATE_KEY	128 (&h80)	Status-Taste (Shift, Strg, Alt ...)
KF_EXTENDED	16 (&h10)	"Erweiterte" Taste (abgesetzte Steuertaste) (Cursor, Einfg, Pos1 ...)
—	8 (&h08)	temporäre Accent-Taste
KF_FIRST_PRESS	4 (&h04)	Taste wurde gerade frisch gedrückt
KF_REPEAT_PRESS	2 (&h02)	Taste gehalten, Autorepeat Funktion
KF_RELEASE	1 (&h01)	Taste wurde losgelassen

inputFlags

Mit Hilfe der Instancevariablen inputFlags kann man steuern, ob ein Tastaturereignis vom Objekt selbst, vom BASIC Handler oder von beiden behandelt werden soll.

Syntax UI-Code: **inputFlags** = **bits**
bits: Kombination der IF_-Werte laut Tabelle unten
Lesen: **<numVar>** = **<obj>**.**inputFlags**
Schreiben: **<obj>**.**inputFlags** = **bits**

Per Default ist inputFlags Null, d.h. Tastaturereignisse werden sowohl vom Objekt als auch vom BASIC-Handler (so einer gesetzt ist) behandelt. Als Faustregel gilt: Setzen Sie nur die Bits, die sie auch wirklich für die Programmfunktion benötigen.

Die folgenden Werte bzw. Konstanten sind definiert:

Konstante	Wert	hexadezimal
IF_IGNORE_FIRST_PRESS	1	&h01
IF_IGNORE_REPEAT_PRESS	2	&h02
IF_IGNORE_RELEASE	4	&h04
IF_IGNORE_ANY_KEY	7	(7 = 1 + 2 + 4)
IF_FILTER_GENERATED_EVENTS	8	&h08
IF_HANDLER_NO_FIRST_PRESS	16	&h10
IF_HANDLER_NO_REPEAT_PRESS	32	&h20
IF_HANDLER_NO_RELEASE	64	&h40
IF_HANDLER_GENERATED_EVENTS	128	&h80
IF_DONT_MAP_NUM_PAD	256	&h100
IF_MAPPED_NUM_PAD_STATE_BIT	512	&h200

IF_IGNORE_FIRST_PRESS
IF_IGNORE_REPEAT_PRESS
IF_IGNORE_RELEASE
IF_IGNORE_ANY_KEY

Diese Bits **verhindern**, dass **das Objekt** die entsprechenden Ereignisse selbst behandelt bzw. an seine Children / sein Content weiterleitet. Das betrifft jedoch nur "echte" Tastendrücke. Tastaturereignisse, die mit der Methode **KbdEvent** erzeugt wurden, werden immer vom Objekt behandelt bzw. weitergeleitet.

IF_FILTER_GENERATED_EVENTS

Dieses Bit bewirkt, dass die Bits IF_IGNORE_FIRST_PRESS, IF_IGNORE_-REPEAT_PRESS, IF_IGNORE_RELEASE und IF_IGNORE_ANY_KEY auch auf Ereignisse wirken, die mit der Methode **KbdEvent** erzeugt wurden. Das Bit IF_FILTER_GENERATED_EVENTS wird sehr selten gebraucht.

IF_HANDLER_NO_FIRST_PRESS
IF_HANDLER_NO_REPEAT_PRESS
IF_HANDLER_NO_RELEASE

Diese Bits **verhindern**, dass **der BASIC Handler** die entsprechenden Ereignisse behandeln kann.

IF_HANDLER_GENERATED_EVENTS

Per Default wird der BASIC Handler für Tastaturereignisse, die mit KbdEvent erzeugt wurden, nicht gerufen. Dieses Bit aktiviert den Handler für solche Ereignisse. Die IF_HANDLER_NO_~ Bits werden dabei berücksichtigt.
Das Bit IF_HANDLER_GENERATED_EVENTS wird sehr selten gebraucht.

IF_DONT_MAP_NUM_PAD
IF_MAPPED_NUM_PAD_EXT_BIT

Aus historischen Gründen senden die Ziffern und Operatorzeichen vom abgesetzten Ziffernblock eigentlich erweiterte Tastencodes. Das erschwert die Auswertung der Tasten jedoch sehr. Deshalb wandelt R-BASIC diese Tastencodes intern in "normale" Codes um, so dass für den BASIC Keyboardhandler kein Unterschied besteht. Das Flag IF_MAPPED_NUM_PAD_STATE_BIT weist R-BASIC an, für diese Tastencodes das Bit KF_EXTENDED zu setzen, so dass Sie diese Tasten wieder von den "normalen" Tasten unterscheiden können. Das Bit IF_DONT_MAP_NUM_PAD schaltet die Umwandlung komplett aus. Damit haben Sie die volle Kontrolle aber auch sehr viel mehr Aufwand.

Beispiel: Die Cursortasten sollen die Spielfigur Willy steuern.

Das Tastaturhandling wird von einem BitmapContent gemacht. Da dieses keinen eigenen Tastaturhandler und keine Children hat benötigen wir kein IF_IGNORE~ Bits. Außerdem wollen wir bei jedem Tastendruck nur genau einen Schritt machen, auch wenn die Taste länger gedrückt ist. Deswegen leiten wir REPEAT_PRESS und RELEASE-Ereignisse nicht an den BASIC Handler weiter. Die SUB's MoveWillyUp usw. müssen natürlich irgendwo definiert sein.

UI-Code

```
BitmapContent GameBoard
    OnKeyPressed = KeyHandler
    inputFlags = IF_HANDLER_NO_REPEAT_PRESS + \
                 IF_HANDLER_NO_RELEASE
End Object
```

BASIC Code

```
Include "KeyCodes"          ' Konstanten KEY_UP usw. einbinden

KEYBOARDACTION KeyHandler
   ON character SWITCH
   CASE KEY_UP:             ' Cursor nach oben
       MoveWillyUp
       END CASE
   CASE KEY_DOWN:           ' Cursor nach unten
       MoveWillyDown
       END CASE
   CASE KEY_LEFT:           ' Cursor nach links
       MoveWillyLeft
       END CASE
   CASE KEY_RIGHT:          ' Cursor nach rechts
       MoveWillyRight
       END CASE
   END SWITCH
End ACTION
```

14.3 Simulieren von Tastaturereignissen

Methode	Aufgabe
KbdEvent	Erzeugt ein Tastaturereignis (Scancode ist Null)
KbdEventWithScancode	Erzeugt ein Tastaturereignis mit Scancode

Die Methoden **KbdEvent** und **KbdEventWithScancode** erzeugen ein Tastaturereignis und senden es an das entsprechende Objekt. Diese Methoden sind für **alle Objektklassen** definiert.
In den meisten Fällen reicht es aus, **KbdEvent** zu verwenden.

KbdEvent

Syntax im BASIC Code: **<obj>.KbdEvent character, keyState, keyFlags**
character: ASCII-Code oder erweiterter ASCII-Code
keyState: Kombination von KS_~Bits (siehe OnKeyPressed)
keyFlags: Eines der KF_~ Bits (siehe OnKeyPressed)

Die übergebenen Werte character, keyState und keyFlags werden 1:1 als Parameter an den Tastaturhandler des Objekts und an den BASIC Tastaturhandler weitergegeben. Im Unterschied zum echten Tastendruck übergibt KbdEvent als "scanCode" jedoch den Wert Null an das Objekt, so dass sowohl das Objekt als auch der BASIC Handler echte von vorgetäuschten Tastendrücken unterscheiden können. Per Default werden mit KbdEvent simulierte Tastaturereignisse **nicht vom Objekt** behandelt, sondern sie werden gleich an den BASIC Handler weitergegeben. Details dazu siehe auch: inputFlags.

Beispiel 1: Kopieren der Tastatureingaben an ein zweites Textobjekt
UI-Code

```
Memo Text1
  OnKeyPressed = KeyHandler
End OBJECT

Memo Text2
End OBJECT
```

BASIC Code

```
KEYBOARDACTION KeyHandler
  ' Handler von Text1 sendet an Text2
  Text2.KbdEvent character, keyState, keyFlags
End ACTION
```

Beispiel 2: Senden eines Zeichens an das Application-Objekt. Das Ereignis wird dann intern an das Objekt, das den Focus hat, weitergeleitet.

```
DemoApplication.KbdEvent ASC("A"), 0, KF_FIRST_PRESS
DemoApplication.KbdEvent ASC("A"), 0, KF_RELEASE
```

Beispiel 3: Senden eines Zeichens an das Target-Objekt.

```
Target.KbdEvent ASC("Z"), 0, KF_FIRST_PRESS
Target.KbdEvent ASC("Z"), 0, KF_RELEASE
```

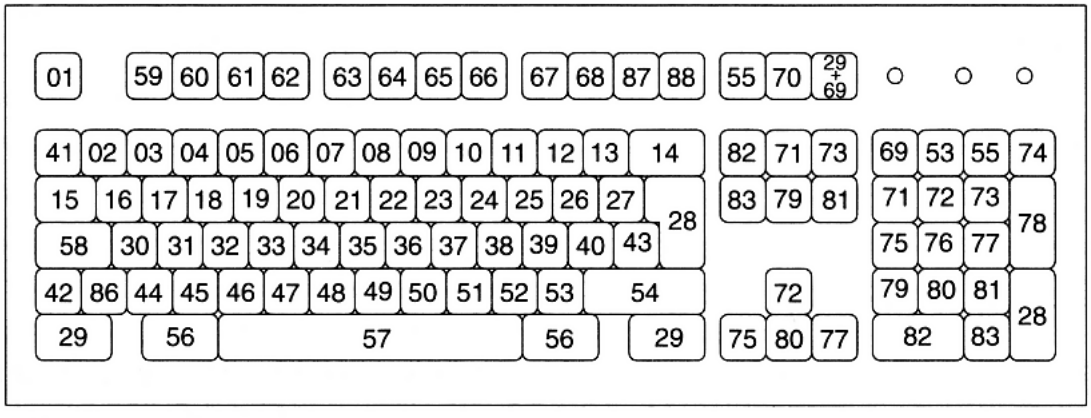
KbdEventWithScancode

In einigen Fällen werten die Objekte außer dem ASCII-Code und dem Tastaturstatus auch den Scancode der Taste aus. Das ist insbesondere bei der Tastaturnavigation durch Menüs der Fall, wenn bei der Definition des Tastenkürzels (siehe Instancevariable kbdShortcut, Objekthandbuch, Kapitel 3.1.4) das Bit KSM_PHYSICAL gesetzt ist. Typischer Weise ist dieses Bit bei den Menüeinträgen zum Drucken (Strg-P), Kopieren (Strg-C), Einfügen (Strg-V) usw. gesetzt.
Für den seltenen Fall, dass Sie einen solchen Button durch ein simuliertes Tastaturereignis aktivieren wollen gibt es die Methode KbdEventWithScancode. In allen anderen Fällen sollten Sie die Methode KbdEvent verwenden.

Syntax im BASIC Code:
<obj>.KbdEventWithScancode character, keyState, keyFlags, scanCode

character:	ASCII-Code oder erweiterter ASCII-Code
keyState:	Kombination von KS_~Bits (siehe OnKeyPressed)
keyFlags:	Eines der KF_~ Bits (siehe OnKeyPressed)
scanCode:	Scancode der zu 'character' gehörenden Taste

Die übergebenen Werte werden 1:1 als Parameter an den Tastaturhandler des Objekts und an den BASIC Tastaturhandler weitergegeben. Im Unterschied zu KbdEvent muss der Scancode der simulierten Taste übergeben werden. Das Objekt kann jetzt den simulierten Tastendruck nicht mehr von einem echten Tastendruck unterscheiden.
Passen ASCII-Code und Scancode nicht zueinander, so erkennen Objekte, die den Scancode auswerten, die Taste nicht. Das folgende Bild zeigt die Scancodes einer MF II Tastatur. Diese Codes sind auch für neuere Tastaturen gültig, die weitere Tasten (und damit weitere Scancodes) haben.



Scancodes einer MF II Tastatur

14.4 Komplexes Beispiel - Filtern von Tastaturereignissen

Wir wollen bei einem Textobjekt nur Ziffern zulassen sowie die Buchstaben im Bereich von 'A' bis 'F'. Das Objekt soll alle anderen Buchstaben und Zeichen herausfiltern. Ein Blick auf die Instancevariable textFilter ergibt, dass es dafür keinen passenden Wert gibt.

Diese Aufgabe erfordert eine komplexe Lösung, da das Objekt per Default das Tastaturereignis zuerst selbst behandelt. Im Anschluss daran ruft es den BASIC Handler. Wir wollen jedoch, dass der BASIC Handler das Ereignis zuerst erhält, damit wir unerwünschte Zeichen herausfiltern können. Erst danach darf das Textobjekt das Ereignis behandeln.

Um das zu erreichen muss man die inputFlags auf den Wert IF_IGNORE_ANY_KEY setzen und einen Keyboardhandler implementieren.

UI Code

```
Memo Text1
  OnKeyPressed = KeyHandler
  inputFlags = IF_IGNORE_ANY_KEY       ' betrifft nur echte
                                       ' Tastendrücke
End OBJECT
```

Intern passiert jetzt folgendes:

1. Das Textobjekt erhält ein Tastaturereignis. Es prüft die inputFlags und den scanCode und stellt fest:
   - Ereignis nicht selbst behandeln
   - BASIC Handler rufen
2. Der BASIC Handler erhält das Tastaturereignis. Er prüft den Tastencode und entscheidet, ob er den Tastendruck ignorieren oder an das Textobjekt zurückgeben soll. In folgenden Fällen soll der Tastendruck an das Textobjekt zurückgegeben werden:
   - Es ist ein Steuerzeichen oder eine Sondertaste.
   - Es ist ein erwünschtes ASCII-Zeichen (0 ... 9 und A ... F)

   Um das Ereignis an das Objekt zurückzusenden verwenden wir die Methode KbdEvent.
3. Das Textobjekt erhält das vom BASIC-Handler "künstlich" erzeugte Tastaturereignis. Das Objekt erkennt automatisch, dass es sich um einen "unechten" Tastendruck handelt (weil der Parameter scanCode = 0 ist) und behandelt es deshalb jetzt, ohne den BASIC-Handler noch einmal zu rufen.

Im OnKeyPressed-Handler müssen wir unbedingt dafür sorgen, dass alle Steuertasten, Shift, Alt usw. durchgereicht werden. Sonst funktionieren die Keyboard Shortcuts und schlimmstenfalls die gesamte Tastaturnavigation nicht mehr. Ausnahmen wären Tastencodes, die wir bewusst selbst verwenden. Beispielsweise könnten wir die Entertaste herausfiltern. In unserem Fall lassen wir alle erweiterten Tasten passieren und blocken nur diejenige darstellbaren ASCII-Codes, die uns nicht interessieren. Bei der Gelegenheit wandeln wir auch gleich Kleinbuchstaben in Großbuchstaben um.

BASIC Code

```
! -------- Handler KeyHandler --------
! Aufgabe:           Tastaturereignisse ausfiltern
! Parameter: sender AS Object, character AS Word,
!      keyState AS Word, keyFlags AS Byte, scanCode AS Byte
!     scanCode = Null wenn mit Methode KbdEvent erzeugt
! -------------------------------
KEYBOARDACTION KeyHandler
DIM ch

   ' Statt den Befehl GOTO zu verwenden erzeugen wir
   ' eine "Endlos-Schleife"
   ' Diese verlassen wir mit BREAK, wenn wir einen
   ' akzeptablen ASCII-Code gefunden haben. Dann wird
   ' hinter der Schleife weiter gemacht.
   ' Wir verlassen sie mit RETURN, wenn wir den ASCII-Code
   ' ignorieren wollen

REPEAT

   ' Steuerzeichen und Sondertasten haben die höherwertigen
   ' Bits im Parameter "character" gesetzt.
   ' Wir verlassen in diesem Fall die Schleife mit BREAK.
   IF character AND &hFF00 THEN BREAK

   ' Normale Codes prüfen
   ch = character                ' Das ist kürzer zu schreiben
                                 ' aber eigentlich überflüssig
   ' Ziffer ?
   IF (ch >= ASC("0") ) AND (ch <= ASC("9")) THEN BREAK

   ' Buchstabe von A bis F ?
   IF (ch >= ASC("A") ) AND (ch <= ASC("F")) THEN BREAK

   ' Kleinbuchstabe von a bis f ?
   IF (ch >= ASC("a") ) AND (ch <= ASC("f")) THEN
      character = ch - 32        ' a->A usw.
      BREAK                      ' Buchstabe
      End IF

   RETURN                        ' Code nicht akzeptieren
   UNTIL TRUE


  ' Jetzt ASCII-Code an das Textobjekt zurücksenden
  sender.KbdEvent character, keyState, keyFlags

END ACTION
```

Den kompletten Quellcode hierfür finden Sie bei den Beispielen unter "Objekte\Text\Keyboard Handler Demo".

(Leerseite)

15 Implementieren eines Dokument-Interfaces

Im PC/GEOS SDK wird die Arbeit mit Dokumenten durch drei Objektklassen realisiert. Diese Klassen erzeugen die benötigte UI selbst und arbeiten eng zusammen. Dazu müssen bestimmte Messages in bestimmter Weise gehandelt werden. Das ist so in R-BASIC nicht realisierbar. Deswegen muss sowohl die UI als auch die Routinen mit BASIC-Mitteln nachgebildet werden.

Dieses Kapitel beschreibt, wie man zur Realisierung eines Dokument-Interfaces unter R-BASIC vorgehen muss. Alle hier beschriebenen Routinen und UI-Objekte sind dem Beispiel "Dokument Interface" entnommen, das sich im Ordner "R-BASIC\Beispiel\Objekte\Dateiarbeit" befindet. Dort finden Sie auch die hier aus Platzgründen nicht aufgeführten Objekte.

Um ein eigenes Dokument-Interface zu erstellen sollten Sie in R-BASIC im Menü "Extras" den Punkt "Code-Sequenz" -> "Dokument-Interface" verwenden. Dort werden die in diesem Kapitel beschriebenen Routinen bzw. UI-Objekte bereitgestellt und automatisch in Ihren Code eingefügt. Dabei können Sie auswählen, ob Sie das komplette hier beschriebene Interface implementieren wollen oder nur Teile davon. Wenn Sie einmal verstanden haben, wie das Prinzip geht, können Sie auch leicht weitere Features, die hier nicht besprochen sind, hinzufügen.

15.1 Konzeptionelles

Unser Dokument-Interface soll folgendes leisten:

Standard-Dateioperationen:
- Neu, Öffnen, Schließen, letzter Stand, Speichern, Speichern unter
- Quick-Backup: Backup anlegen und aus Backup wiederherstellen.
- Muster: Als Muster speichern, Muster öffnen
- Verschieben nach ... , Kopieren nach ..., Umbenennen
- Bearbeiten der Benutzernotizen
- Die Funktionen "Import" und "Benutzerebene ändern" werden zur Demonstration vorbereitet, aber nicht implementiert.

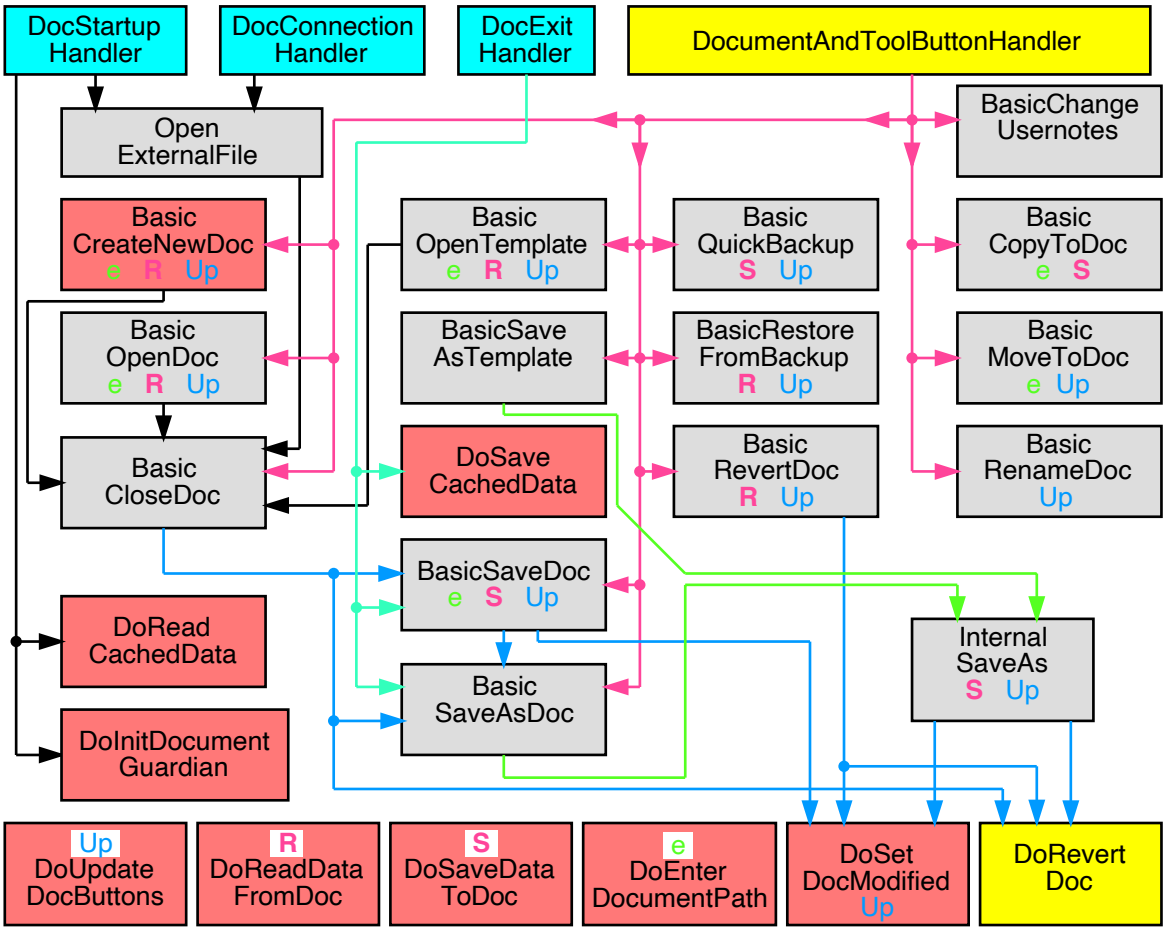
Fähigkeiten:
- Arbeit mit DOS- und GEOS-Dateien (auch mit schreibgeschützten) ist möglich
- Wenn die aktuelle Datei ungeändert ist kann man "Neu" und "Öffnen" anwählen ohne die aktuelle Datei vorher schließen zu müssen.
- Es kann immer nur eine Datei gleichzeitig offen sein.
- Das Programm kann darauf reagieren, wenn der Nutzer eine verknüpfte Datei doppelklickt. Für DOS-Dateien muss der Nutzer dazu das Programm über einen entsprechenden Eintrag in der GEOS.INI mit einer Dateierweiterung verknüpfen. Für GEOS Dateien setzt das Programm das Token und das Creator-Token der Datei automatisch.
- Jedes Mal, wenn eine Applikations-spezifische Operation nötig ist, z.B. Daten in die Datei schreiben, wird stattdessen eine MessageBox aufgerufen.
- Ein offenes Dokument soll einen Systemrestart "überleben".

Software Entscheidungen:

- Alle Buttons aus dem Datei-Menü und aus der Toolbar bekommen einen actionData-Wert, der ihre Funktion beschreibt. Ein einziger, zentraler Action-Handler (**DocumentAndToolButtonHandler**), der von allen Buttons aktiviert wird, ruft dann die entsprechenden Routinen auf.
- Eine einzige zentrale Routine (**DoUpdateDocButtons**) ist dafür zuständig die zum Dokument-Interface gehörenden Buttons an den aktuellen Zustand des Dokuments anzupassen.
- Änderungen werden erst dann in die Datei geschrieben, wenn der Nutzer dies explizit anweist. In diesem Kontext bedeutet der Terminus "das Dokument wurde geändert", dass der Nutzer die Daten im Programm verändert hat. Er bedeutet **nicht**, dass die Änderungen schon in die Datei geschrieben wurden! Wenn Sie VM-Dateien verwenden dürfen Sie von diesem Prinzip abweichen.

Grundstruktur des Dokument-Interface

Das folgende Bild zeigt alle Routinen, die zur vollständigen Implementation der oben genannten Fähigkeiten notwendig sind.



Die unten links angeordneten Routinen werden an vielen Stellen aufgerufen. Die Zuordnung erfolgt deshalb nicht durch Pfeile, sondern durch die in den Kästchen vermerkten Buchstaben. Zum Beispiel rufen alle Routinen, die ein "R" im Kästchen haben die Routine DoReadDataFromDoc.

Im Bild sind drei Gruppen von Routinen zu sehen:

- Routinen, die in Cyan unterlegt sind, sind ActionHandler. Sie werden vom System gerufen und müssen als die passenden Actionhandler in das Application-Objekt eingebunden werden.
- Routinen, die in Rot unterlegt sind, sind Programmspezifisch. Sie **müssen** diese Routinen anpassen bzw. erweitern um die Funktionen Ihres Programms zu realisieren.
- Ob die Routinen, die in Gelb unterlegt sind, geändert / angepasst werden müssen hängt von der Komplexität Ihres Programms ab.
- Routinen, die in Grau unterlegt sind, enthalten allgemeingültigen Code. Im Normalfall ist es nicht notwendig diese Routinen anzupassen.

Bei der Verwaltung von Dokumenten fallen eine Reihe von Aufgaben an, die unabhängig von der Art und der Struktur der eigentlichen Dokumentdatei sind. R-BASIC unterstützt das durch die Bereitstellung einer Objektklasse und einer Library. Beide arbeiten eng zusammen.

Die Objektklasse **DocumentGuardian** erleichtert Ihnen den Umgang mit Dokumenten, indem Sie allgemeine Informationen, die bei der Arbeit mit Dokumenten anfallen, verwalten. Dazu zählen z.B. der Name und der Pfad zur Dokumentendatei sowie das FileHandle der offenen Datei. Außerdem können Objekte der Klasse DocumentGuardian vorhandenen Dokumente öffnen, neue Dokumente anlegen und offene Dokumente schließen. Dabei berücksichtigen sie z.B. den Dateityp und das Token, behandeln schreibgeschützte Dateien korrekt und vieles mehr. Auf diese Weise entlasten diese Objekte den BASIC-Programmierer von einer Vielzahl von Standardaufgaben.

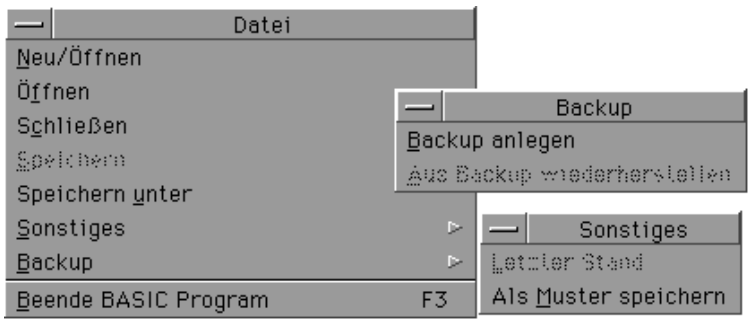
Die Library **DocumentTools** stellt eine Reihe von Funktionen bereit, die Sie bei der Arbeit mit Dokumenten unterstützen. Dazu gehören zum Beispiel die typischen Dialogboxen zum Öffnen oder Speichern einer Datei. Um die DocumentTools Library nutzen zu können müssen Sie sie ihn Ihr Programm einbinden:

```
Include "DocumentTools"
```

Die DocumentTools Library muss separat heruntergeladen werden, sie ist nicht Teil des R-BASIC Standard-Pakets. Wenn Sie mit VM-Dateien als Dokumente arbeiten wollen müssen Sie außerdem die VMFiles Library herunterladen und includen.

Um die in diesem Kapitel beschriebenen Funktionen und Zusammenhänge zu verstehen sollten Sie zumindest den einführenden Abschnitt zum Document-Guardian-Objekt (Objekt Handbuch, Kapitel 4.13) sowie das Kapitel 1 (Konzeptionelles) aus dem Handbuch der DocumentTools Library gelesen haben.

15.2 Die UI: Datei-Menü und Toolbar



Das Dateimenü und die Toolbar bestehen aus einer Reihe von Buttons. Die Grafiken für die Buttons finden wir im Ordner "Icon Tool Graphics\Document", einem Unterordner von "USERDATA\R-BASIC\IMAGES".

Jeder Button hat als ActionHandler den zentralen Handler **DocumentAnd-ToolButtonHandler** gesetzt und der actionData-Wert bestimmt die Funktion des Buttons. Aus Platzgründen sind nur einige der Objekte gezeigt.

```
Group DocumentMenuButtons
   Children = MenuNewOpenButton, MenuOpenButton, MenuCloseButton,
              MenuSaveButton, MenuSaveAsButton, MenuBackupGroup,
              MenuOtherGroup
End OBJECT

Button MenuNewOpenButton
  Caption$ = "Neu/Öffnen ",  0
  ActionHandler = DocumentAndToolButtonHandler
  actionData = ID_NEW_OPEN_DIALOG
  BringsUpWindow
End OBJECT
Button MenuOpenButton
  Caption$ = "Öffnen " , 1
  ActionHandler = DocumentAndToolButtonHandler
  actionData = ID_OPEN_DOC
  BringsUpWindow
End OBJECT

<...>
```

Einzige Besonderheit im UI-Code der Group DocumentToolButtons ist der Hint ExpandHeight bei der Group und den Buttons selbst, der bewirkt, dass sich die Buttons vergrößern, wenn sie gemeinsam mit größeren Objekten in der Group DocumentToolGroup verwendet werden.

```
Group DocumentToolGroup
  Children = DocumentToolButtons
  ExpandWidth
End OBJECT
```

```
Group DocumentToolButtons
   Children = ToolNewButton, ToolOpenButton, ToolTemplateButton,
              ToolCloseButton, ToolSaveButton, ToolBackupButton
  OrientChildren = ORIENT_HORIZONTALLY
  MakeToolbox
  ExpandHeight
End OBJECT
Button ToolNewButton
  CaptionImage = "Icon Tool Graphics\\Document\\NEW.GIF"
  ActionHandler = DocumentAndToolButtonHandler
  actionData = ID_NEW_DOC
  ExpandHeight
End OBJECT
Button ToolOpenButton
  CaptionImage = "Icon Tool Graphics\\Document\\OPEN.GIF"
  ActionHandler = DocumentAndToolButtonHandler
  actionData = ID_OPEN_DOC
  ExpandHeight
End OBJECT

<...>
```

15.3 Kernroutinen und Tools

Dieser Abschnitt behandelt grundlegende Routinen, die zur Implementation der in den folgenden Abschnitten besprochenen Funktionen benötigt werden. Im Einzelnen sind das

- der Actionhandler **DocumentAndToolButtonHandler**. Im Normalfall müssen Sie an dieser Routine nichts ändern. Nur bei sehr komplexen Programmen kann manchmal nötig sein, sie anzupassen. In diese Gruppe gehört auch die Sub **DoRevertDoc**. Sie wird im Kapitel 15.6 (Letzter Stand) beschrieben.
- die Subs **DoSetDocModified**, **DoInitDocumentGuardian**, **DoUpdateDoc-Buttons**, **DoReadDataFromDoc**, **DoSaveDataToDoc**, **DoReadCachedData**, **DoSaveCachedData** und **DoEnterDocumentPath**. Sie alle müssen angepasst werden um die Funktionen Ihres Programms zu implementieren.

Der zentrale Handler **DocumentAndToolButtonHandler** handelt jeden Klick auf einen Button im Datei-Menü. Um die Buttons zu unterscheiden muss jeder Button einen actionData-Wert gesetzt haben, der seine Funktion beschreibt. Die entsprechenden Konstanten sind in der Library DocumentTools definiert und auch dort beschrieben.
Der Handler ist im Folgenden in Teilen wiedergegeben. In der ON - SWICTH Anweisung werden die Anweisungen entsprechend dem actionData Wert des gedrückten Buttons ausgeführt. Die dort verwendeten Routinen haben häufig selbsterklärende Namen. Sie werden weiter unten ausführlich beschrieben.

Hingewiesen werden soll auf drei Dinge:

1. Im Fall ID_NEW_OPEN_DIALOG wird nur die NewOpenDialogBox geöffnet, danach ist der Handler beendet. R-BASIC läuft dann solange im Leerlauf bis der Nutzer einen Button aus diesem Dialog anklickt. Das ist so gewollt.
2. Die Routine BasicCloseDoc schließt das aktuelle Dokument. Sie erwartet einen Parameter, der bestimmt, ob versucht werden soll, eventuell geänderte Daten zu speichern. Sollte es ein Problem dabei geben liefert BasicCloseDoc TRUE zurück und wir können die gewünschte Aktion abbrechen.
   Im Gegensatz zu den meisten anderen Routinen updatet BasicCloseDoc die UI nicht. Deswegen muss das hier erledigt werden. Der Grund dafür ist, dass BasicCloseDoc von verschiedenen anderen Routinen gerufen wird, z.B. wenn ein neues Dokument geöffnet wird.
3. Wir brauchen keine Abfragen, ob die gewünschte Aktion gerade erlaubt ist. Das erledigt die Routine **DoUpdateDocButtons** für uns, indem sie Buttons, die gerade nicht erlaubte Aktionen ausführen würden, disabled.

```
BUTTONACTION DocumentAndToolButtonHandler
DIM err

ON ACTIONData SWITCH

  CASE ID_NEW_OPEN_DIALOG:
      DTShowNewOpenDialog(ConvertObjForSDK(DocumentObj),
          NOF_NEW_OPEN_TEMPLATE + NOF_CONFIG + NOF_IMPORT, "")
      END CASE
```

```
   CASE ID_NEW_DOC:
       BasicCreateNewDoc
       END CASE

   CASE ID_OPEN_DOC:
       BasicOpenDoc
       END CASE

   CASE ID_CLOSE_DOC:
       err = BasicCloseDoc(TRUE)     ! Änderungen speichern
       IF err THEN RETURN
       DoReadDataFromDoc             ' UI updaten
       DoUpdateDocButtons
       End CASE

   CASE ID_SAVE_DOC:
       BasicSaveDoc
       END CASE

< ... usw. ...>

   END SWICTH

END ACTION
```

Die Routine **DoSetDocModified** ermöglicht die Verwaltung der Information, ob das aktuelle Dokument geändert wurde oder nicht. Das ermöglicht z.B. der Routine BasicCloseDoc den Nutzer zu fragen, ob er die Änderungen speichern möchte oder nicht. Für diesen Zweck verwaltet das DocumentGuardian-Objekt das Bit DOCS_MODIFIED in seiner Instancevariablen documentState. Der Parameter "modi" bestimmt, ob das Dokument als "modifiziert" (modi = TRUE) oder als "nicht modifiziert" (modi = FALSE) gekennzeichnet werden soll. Entsprechend setzt die Methode SetDocumentState dieses Bit oder setzt es zurück. Abschließend wird mit DoUpdateDocButtons die UI angepasst.
Sie sollten die Routine **DoSetDocModified** jedes Mal rufen, wenn der Nutzer Daten des Dokuments ändert. Typische Situationen sind der OnModified-Handler von Text-Objekten, die Apply-Handler der Objektklassen Number, Memo, InputLine sowie der Listenobjekte oder der ColorChangedHandler von ColorSelector-Objekten. Die Routine fragt jeweils selbst ab, ob sich der "modifiziert"-Zustand geändert hat und kehrt sofort zurück, wenn dem nicht so ist.
Wichtig: Nehmen wir an, Sie haben ein Text-Objekt (Memo, InputLine, VisText oder LargeText), und das Dokument soll "modifiziert" sein, wenn der Nutzer etwas eingibt.
Text-Objekte senden ihre OnModified-Message aber nur aus, wenn der Nutzer "erstmalig" etwas eingibt, das heißt wenn sie vom Zustand "nicht modifiziert" in den Zustand "modifiziert" wechseln. Deswegen müssen Sie den "modified" Zustand der betroffenen Textobjekte zurücksetzen, wenn DoSetDocModified mit dem Parameter modi=FALSE gerufen wird.
Der Apply-Handler von Listen- und Number-Objekten wird im Allgemeinen jedes Mal aufgerufen, wenn er Nutzer etwas ändert. Deswegen ist es normalerweise nicht nötig, den Modified-Zustand dieser Objekte hier zu ändern.

```
SUB DoSetDocModified (modi as INTEGER)

  IF  modi THEN
    ' ist schon "modified"? => Return
    IF DocumentObj.documentState AND DOCS_MODIFIED THEN RETURN
    DocumentObj.SetDocumentState DOCS_MODIFIED, 0
  ELSE
    ' ist schon "nicht modified"? => Return
    IF (DocumentObj.documentState AND DOCS_MODIFIED) = 0 THEN
  RETURN
    DocumentObj.SetDocumentState 0, DOCS_MODIFIED
  End IF

  ' -X- Bei Bedarf: weitere Aktionen.
  ' -> den "Modified" Status von Textobjekten (z.B. Memo),
  '    manchmal auch von Number und Listen-Objekten anpassen
  ' Sie sollten hier NICHT DoReadDataFromDoc rufen
  DoUpdateDocButtons

END SUB
```

Die Sub **DoInitDocumentGuardian** initialisiert das DocumentGuardian-Objekt mit den Daten, die Ihrem Programm entsprechen. Im Beispiel soll das Document-Guardian-Objekt eine GEOS Datendatei (fileType = GFT_DATA) mit dem Token "PHO2", 5 und dem CreatorToken "PHON",5 verwalten. Es ist wichtig, dass das hier angegeben Token mit dem DocToken-Statement und das CreatorToken mit dem AppToken-Statement des Application-Objekts im UI-Code übereinstimmen.
Mit der Anweisung "guardian.ConfigData = dc" werden die Einstellungen an das DocumentGuardian-Objekt übertragen.
Der ButtonHandler wird von der DocumentTools-Library für den "Neuen/Öffnen" Dialog benötigt.

```
SUB DoInitDocumentGuardian(guardian as object)
DIM dc AS DocumentConfigStruct

  guardian.ButtonHandler = DocumentAndToolButtonHandler

  dc.noDocumentString$ = "kein Dokument"
  dc.templateFolder$ = "DemoTemplates"
  dc.nameForNew$ = "Unbenanntes Dokument "

  dc.fileType = GFT_DATA
  dc.creatorToken.tokenChars = "PHON"
  dc.creatorToken.manufid = 5
  dc.token.manufid = 5
  dc.token.tokenChars = "PHO2"
  dc.matchFlags = DOC_MATCH_TOKEN

  guardian.ConfigData = dc

END SUB
```

Eine der wichtigsten Routinen ist die Sub **DoUpdateDocButtons**. Sie hat die Aufgabe, die Dokument-Buttons aus dem Datei-Menü und aus der Dokument-Toolbar sowie ihre eigene UI entsprechend der aktuellen Situation zu enablen oder zu disablen. Dadurch sorgt diese Routine dafür, dass einzelne Funktionen nur dann aufgerufen werden können, wenn sie zulässig sind. Zum Beispiel wird der "Speichern" Button disabled, wenn keine Datei offen ist. Im Allgemeinen müssen Sie diese Routine ergänzen. Ein typischer Fall ist, dass Sie UI Objekte haben (z.B. ein Memo-Objekt), in die der Nutzer Daten für das aktuelle Dokument eingeben kann. Diese UI muss disabled werden, wenn kein Dokument offen ist.

Die Entscheidung, ob ein Button enabled sein soll oder nicht kann komplex und unübersichtlich sein. Deswegen stellt die DocumentTools Library die Routine DTFindEnabled bereit, die genau diese Information liefert. Welcher Button gemeint ist wird über eine Konstante bestimmt, die in der Library definiert ist. Sie wird auch für den actionData-Wert des Buttons benutzt wird. Der folgende Code zeigt die Routine in Ausschnitten. DTFindEnabled erwartet eine Referenz auf das DocumentGuardian-Objekt, die mit der BASIC-Routine ConvertObjForSDK konvertiert wurde. Die Variable docObj speichert die konvertierte Referenz auf das Objekt.
Die Anweisung "DemoPrimary.Caption2$ = DocumentObj.documentName$" bewirkt, dass der Name des aktuell offenen Dokuments in der Titelzeile des Primary-Objekts angezeigt wird. Hier müssen Sie den Namen ihres eigenen Primary-Objekts verwenden.

```
SUB DoUpdateDocButtons ()
DIM docObj as Object
DIM docState

    docObj = ConvertObjForSDK(DocumentObj)

    MenuNewOpenButton.enabled = DTFindEnabled(docObj,
             ID_NEW_OPEN_DIALOG)
    MenuOpenButton.enabled = DTFindEnabled(docObj, ID_OPEN_DOC)
    MenuSaveAsButton.enabled = DTFindEnabled(docObj,
             ID_SAVE_AS_DOC)
    MenuCloseButton.enabled = DTFindEnabled(docObj, ID_CLOSE_DOC)
    ToolNewButton.enabled = DTFindEnabled(docObj, ID_NEW_DOC)
    ToolOpenButton.enabled = DTFindEnabled(docObj, ID_OPEN_DOC)
    ToolCloseButton.enabled = DTFindEnabled(docObj, ID_CLOSE_DOC)

< .. usw ...>

    DemoPrimary.Caption2$ = DocumentObj.documentName$

< .. hier eigene UI enablen/disablen ...>

END SUB
```

Die Sub **DoReadDataFromDoc** hat die Aufgabe die im Dokument gespeicherten Daten auszulesen und darzustellen. Weil das immer vom konkreten Programm abhängt erinnert uns eine MsgBox daran, dass wir hier noch etwas programmieren müssen. Wichtig ist, dass wir den Fall "Kein Dokument offen" ebenfalls berücksichtigen. Das können wir durch Abfrage der Instancevariablen documentState auf Null oder wie im Beispiel durch Abfrage der Instancevariablen documentHandle auf NullFile() tun. Häufig werden hier UI-Objekte "geleert", Listen mit Null Einträgen versehen, Texte gelöscht usw.

```
SUB DoReadDataFromDoc ()

   IF DocumentObj.documentHandle == NullFile() THEN
     MsgBox "DoReadDataFromDoc: Keine Datei offen. UI anpassen."
   ELSE
     MsgBox "DoReadDataFromDoc: Hier Daten aus der Datei lesen und
                                      zugehörige UI anpassen."
   End IF

END SUB
```

Die Routine **DoSaveDataToDoc** wird jedes Mal gerufen, wenn Daten dauerhaft in der Datei gespeichert werden sollen. Für VM-Dateien heißt das, dass VMSave gerufen werden muss. In welche Datei die Daten geschrieben werden sollen wird durch den übergebenen Parameter fh bestimmt. Sie dürfen hier NICHT auf DocumentObj.documentHandle zurückgreifen, weil die Routine je nach Kontext (z.B. erstellen eines Backups) auch für andere Dateien gerufen wird.

```
SUB DoSaveDataToDoc (fh as FILE)
  MsgBox "DoSaveDataToDoc: Hier Daten in der Datei fh speichern"
  ' bei Bedarf: FileCommit (fh)
END SUB
```

Beim Herunterfahren von GEOS müssen Dokument-Daten, die in globalen Variablen zwischengespeichert sind, an einem sicheren Platz abgelegt werden. Ein sicherer Speicherplatz für globalen Variablen sind Instancevariablen von Objekten. Diese werden vom System automatisch gesichert. Wenn man genau eine globale Struktur hat bietet sich die Instancevariable privData des Document-Guardian-Objekts dafür geradezu an.
Die Routine **DoSaveCachedData** wird vom OnExit-Handler gerufen und muss die globalen Variablen sichern. Das Gegenstück **DoReadCachedData** wird vom OnStartup Handler gerufen und muss die globalen Variablen wieder herstellen.
Für den Fall, dass Sie keine Dokument-Daten in globalen Variablen zwischen-speichern, können Sie diese Routinen löschen.

```
SUB DoSaveCachedData (docObj as Object)
  ' Globals Struktur in "privData" speichern
  ' -X- docObj.privData = globalData, sizeof(GlobalDataStruct)
END SUB 'DoSaveCachedData
```

```
SUB DoReadCachedData (docObj as Object)
    ' DoReadCachedData: Hier globale Variablen wiederherstellen.
    ' Die UI muss nicht upgedatet werden
  ' -X- globalData = docObj.privData
END SUB 'DoReadCachedData
```

Das Tool **DoEnterDocumentPath** wählt den Pfad an, in dem die Dokumente gespeichert werden sollen. Weil das von Ihrem Programm abhängt müssen Sie diese Routine ändern. Wenn der Parameter forNew TRUE ist wird der Pfad angewählt, in dem neu angelegte Dokumente gespeichert werden sollen. Häufig ist das der GEOS-Top-Ordner (SP_TOP). Wenn der forNew FLASE ist wählt die Routine den Ordner an, in dem standardmäßig benannte Dokumente abgelegt werden. Die Anweisung CreateDir stellt sicher, dass der Ordner existiert. CreateDir hat kein Problem damit, wenn der Ordner bereits existiert. Sie können die letzten beiden Statements auskommentieren um den Dokument-Ordner direkt zu verwenden.

```
SUB DoEnterDocumentPath (forNew as Integer)

   IF forNew THEN
       SetStandardPath SP_TOP
   ELSE
       SetStandardPath SP_DOCUMENT
       CreateDir "Subfolder"
       SetCurrentPath "Subfolder"
   END IF

END SUB
```

15.4 Standard Dokument Operationen

Alle für die Dateioperationen notwendigen Dialoge werden von der Document-Tools Library bereitgestellt. Die entsprechenden Routinen liefern eine Struktur zurück, die folgendermaßen definiert ist:

```
STRUCT DialogReturnStruct
  fileName$ as String[32]
  retInfo As Word
  End Struct
```

Das Feld "fileName$" enthält den vom Nutzer eingegeben bzw. ausgewählten Dateinamen. Das Feld "retInfo" enthält die Information, welchen Button im Dialog der Nutzer gedrückt hat. Dafür dien folgenden Konstanten definiert:

Konstante	Wert	Bedeutung
DRI_CANCEL	1	Der Nutzer hat "Abbrechen" gewählt.
DRI_OK	2	Der Nutzer hat "Speichern", "Öffnen" oder ähnliches gewählt.
DRI_READ_ONLY	3	Der Nutzer hat "Öffnen" oder ähnliches gewählt, die ausgewählte Datei ist jedoch schreibgeschützt oder soll schreibgeschützt geöffnet werden.

15.4.1 Ein neues Dokument anlegen

Klickt der Nutzer im Menü auf den entsprechenden Button so wird im Document-AndToolButtonHandler die Routine BasicCreateNewDoc gerufen.

Um ein neues, leeres Dokument anzulegen muss BasicCreateNewDoc folgende Schritte durchführen:

1. Schließen eines eventuell noch offenen Dokuments. BasicCloseDoc(TRUE) erledigt alle dafür nötigen Schritte, einschließlich der Nachfrage beim Nutzer, ob eventuelle Änderungen gespeichert werden sollen.
2. Wechseln in den Ordner, in dem die neue Datei angelegt werden soll.
3. Anlegen der Datei. Alle dafür nötigen Schritte erledigt die Methode CreateNewDocument. Das DocumentGuardian-Objekt kennt sowohl den Dateityp als auch das Token der Datei und initialisiert die neue Datei entsprechend. VM-Dateien werden so initialisiert, dass sie mit den Routinen aus der VMFiles Library verwendet werden kann.
4. Für den (extrem unwahrscheinlichen) Fall, dass es beim Anlegen der Datei ein Problem gegeben hat, geben wir eine Fehlermeldung aus und verlassen die Sub.
5. Die Datei initialisieren. Hier schreibt man die Daten in die Datei, die auch bei Leeren Dateien vorhanden sein müssen. Ob es da etwas gibt und was das genau sein muss hängt von Ihrem Programm ab.
6. Update der UI mit DoReadDataFromDoc und DoUpdateDocButtons.

```
SUB BasicCreateNewDoc ()
DIM err

  err = BasicCloseDoc(TRUE)
  IF err THEN RETURN

  DoEnterDocumentPath(TRUE)
  DocumentObj.CreateNewDocument
  IF fileError THEN
       MsgBox "Fehler beim Anlegen der neuen Datei. Fehlercode: "
                  + ErrorText$(fileError)
       RETURN
  End IF

  ! Datei initialisieren
  MsgBox "BasicCreateNewDoc: Hier Datei initialisieren."

  DoReadDataFromDoc
  DoUpdateDocButtons

END SUB
```

15.4.2 Öffnen einer Datei

Klickt der Nutzer im Menü auf den entsprechenden Button so wird im Document-AndToolButtonHandler die Routine BasicOpenDoc gerufen.

BasicOpenDoc führt die folgenden Schritte aus:
1. Wechseln in den Pfad, in dem Dokumente normalerweise abgelegt werden.
2. Aufruf der Routine DTOpenDialog aus der DocumentTools Library. Diese Routine zeigt den "Öffnen" Dialog an. Sie liefert eine DialogReturnStruct Struktur zurück (siehe oben). Wenn der Nutzer "Abbrechen gewählt hat verlassen wir die Routine.
3. BasicCloseDoc(TRUE) sorgt dafür, dass eine eventuell noch offene Datei jetzt geschlossen wird. Im Fehlerfall (z.B. wenn der Nutzer "Abbrechen" wählt) verlassen wir die Routine.
4. Öffnen der ausgewählten Datei. Die Methode OpenDocument erledigt alle dazu notwendigen Schritte. Dazu gehört auch, dass schreibgeschützte Dateien schreibgeschützt geöffnet werden. Wird als zweiter Parameter "TRUE" angegeben öffnet die Methode die Datei auf jeden Fall schreibgeschützt.
5. Sollte das Öffnen fehlschlagen geben wir eine Fehlermeldung aus.
6. In allen Fällen updaten wir die UI mit DoReadDataFromDoc und DoUpdate-DocButtons.

```
SUB BasicOpenDoc ()
DIM ret as DialogReturnStruct
DIM err

   DoEnterDocumentPath(FALSE)
   ret = DTOpenDialog(ConvertObjForSDK(DocumentObj), "")
   IF ret.retInfo = DRI_CANCEL THEN RETURN

   err = BasicCloseDoc(TRUE)
   IF err THEN RETURN

   IF ret.retInfo = DRI_OK THEN
        DocumentObj.OpenDocument ret.fileName$
   ELSE
        ' DRI_READ_ONLY, read-only öffnen
        DocumentObj.OpenDocument ret.fileName$, TRUE
   End IF

   IF fileError THEN
        MsgBox "Fehler beim Öffnen der Datei "+ret.fileName$+".
                        Fehlercode: "+ ErrorText$(fileError)
   End IF

   DoReadDataFromDoc
   DoUpdateDocButtons

END SUB
```

15.4.3 Speichern der geänderten Daten

Klickt der Nutzer im Menü auf den Speichern-Button im Menü oder in der Toolbar so wird im DocumentAndToolButtonHandler die Routine BasicSaveDoc gerufen.

BasicSaveDoc dient als Verteiler für die verschiedenen möglichen Fälle:

- Wenn die Datei ungeändert ist ((docState AND DOCS_MODIFIED)=0) kehrt die Routine ohne weitere Aktion zurück. Diese Abfrage greift auch dann, wenn gar keine Datei offen ist (docState = 0).
- Ist die Datei noch unbenannt (docState AND DOCS_UNTITLED ist nicht Null) wird die Datei automatisch unter einem neuen Namen gespeichert. BasicSaveAsDoc() erledigt alle dafür notwendigen Aufgaben.
- Read-Only-Dateien kann man nicht speichern. Deswegen fragen wir den Nutzer ob er die Datei unter einem anderen Namen speichern möchte und starten gegebenenfalls wieder BasicSaveAsDoc().

In allen anderen Fällen rufen wir DoSaveDataToDoc (DocumentObj.documentHandle). Diese Routine erledigt die eigentliche Arbeit. Sie wurde weiter oben (Kapitel 15.3) beschrieben.
Wichtig ist, dass wir den "modified" Status des Dokuments zurücksetzen. Das erledigt die Routine DoSetDocModified(FALSE). Abschließend updaten wir mit DoUpdateDocButtons die Menüs und ggf. andere wichtige UI.

```
SUB BasicSaveDoc ()
DIM docState, ans

   docState = DocumentObj.documentState
   IF (docState AND DOCS_MODIFIED)=0 THEN RETURN

   IF docState AND DOCS_UNTITLED THEN
      BasicSaveAsDoc()
      RETURN
      END IF

   IF docState AND DOCS_READ_ONLY  THEN
      ans = QuestionBox ("Die Datei ist schreibgeschützt. Wollen
                Sie sie unter einem neuen Namen speichern?")
      IF ans = YES THEN
          BasicSaveAsDoc()
      End IF
      RETURN
   End IF

   DoSaveDataToDoc(DocumentObj.documentHandle)
   DoSetDocModified(FALSE)
   DoUpdateDocButtons

END SUB
```

15.4.4 Speichern unter neuem Namen

Klickt der Nutzer im Menü auf den entsprechenden Button so wird im Document-AndToolButtonHandler die Routine BasicSaveAsDoc gerufen. Diese Routine wechselt in den Pfad, in dem Dokumente im Normalfall abgelegt werden und öffnet dann mit DTDaveAsDialog den "Speichern unter" Dialog. Sie ist in der DocumentTools Library definiert und kümmert sich z.B. auch darum, dass der Nutzer nur einen für den vom DocumentGuardian verwalteten Dateityp gültigen Dateinamen eingeben kann. Der Parameter "FALSE" bewirkt, dass die Dateien im FileSelector des "Speichern unter"-Dialogs nicht wie sonst unter GEOS üblich grau angezeigt werden. Dadurch kann der Nutzer z.B. die Datei, die er überschreiben möchte, anklicken.
DTSaveAsDialog wechselt in den Pfad, der im "Speichern unter" Dialog ausgewählt wurde. Sie liefert eine DialogReturnStruct-Struktur zurück (siehe oben), die unter anderem den neuen Namen für die Datei enthält.
Die eigentliche Arbeit erledigt dann die Routine InternalSaveAs, die auch von BasicSaveAsTemplate gerufen wird.

```
FUNCTION BasicSaveAsDoc () AS REAL
Dim err
DIM ret as DialogReturnStruct

  DoEnterDocumentPath(FALSE)

  ret = DTSaveAsDialog(ConvertObjForSDK(DocumentObj), "", FALSE)
  IF ret.retInfo = DRI_CANCEL THEN RETURN TRUE

  err = InternalSaveAs(ret)
  RETURN err

END FUNCTION
```

Sowohl BasicSaveAsDoc als auch InternalSaveAs liefern den Fehlerwert TRUE zurück, wenn der Nutzer den Vorgang abgebrochen hat oder ein anderer Fehler auftrat.

Die Funktion InternalSaveAs erledigt die Hauptarbeit zum Speichern einer Datei unter neuem Namen. Dabei sind die folgenden Schritte zu erledigen:

1. Wir stellen sicher, dass die neue Datei im aktuellen Ordner nicht existiert. Die Routine DTConfirmAndDelete aus der DocumentTools Library prüft das, fragt ggf. den Nutzer, ob er die Datei überschreiben möchte und löscht diese dann. Falls das nicht möglich ist, z.B. weil der Nutzer "Abbrechen" gewählt hat oder weil die Datei in Benutzung ist, liefert dir Routine TRUE zurück und wir verlassen die Routine InternalSaveAs mit dem Returnwert TRUE.
2. Wir fertigen mit DTCloneFile eine 1:1 Kopie des aktuell vom DocumentGuardian geöffneten Dokuments an. Für den sehr unwahrscheinlichen Fall, dass es dabei ein Problem gibt, setzt DTCloneFile die globale Variable fileError und wir brechen den Vorgang ab.
3. Jetzt können wir die aktuelle Datei schließen. Vorher merken wir uns den modified-Status und bringen die Datei mit DoRevertDoc auf den letzten gespeicherten Stand.

4. Nun können wir die neue Datei öffnen. War die Datei geändert speichern wir die den aktuellen Stand in der Datei und setzen den modified-Status zurück.
5. Abschließend bringen wir die Buttons und in den Menüs auf den neuesten Stand und kehren durch "RETURN FALSE" mit der Information "alles OK" zurück.

```
FUNCTION InternalSaveAs (ret as DialogReturnStruct) AS Real
DIM err, modi

  err = DTConfirmAndDelete(ret.fileName$)
  IF err THEN RETURN true

  DTCloneFile(ConvertObjForSDK(DocumentObj), ret.fileName$)
  IF fileError THEN RETURN TRUE

  modi = DocumentObj.documentState AND DOCS_MODIFIED
  DoRevertDoc
  DocumentObj.CloseDocument

  DocumentObj.OpenDocument ret.fileName$
  IF modi THEN
    DocumentObj.SetDocumentState DOCS_MODIFIED, 0
    DoSaveDataToDoc(DocumentObj.documentHandle)
    DoSetDocModified(FALSE)
  END IF

  DoUpdateDocButtons
  RETURN FALSE
END FUNCTION
```

15.4.5 Schließen des Dokuments

Klickt der Nutzer im Menü auf den entsprechenden Button so wird im Document-AndToolButtonHandler die Routine BasicCloseDoc aufgerufen. Außerdem wird BasicCloseDoc an verschiedenen anderen Stellen des Programms gerufen, z.B. wenn eine neue Datei geöffnet werden soll während noch eine andere offen ist.

Die Funktion BasicCloseDoc schließt das aktuelle Dokument und liefert im Erfolgsfall den Wert FALSE zurück. Im Fehlerfall gibt BasicCloseDoc den Fehler-Wert TRUE zurück.

Wenn ein Dokument geschlossen werden soll können vorher andere Aktionen notwendig sein. Zum Beispiel könnte es nötig sein, geänderte Daten in die Datei zu schreiben oder die Datei unter einem anderen Namen zu speichern. Da hier sehr viele Fälle möglich sind bietet die DocumentTools Library die Funktion DTConfirmClose an. Sie prüft die Instancevariable "documentState" des übergebenen DocumentGuardian-Objekts, ob das vom DocumentGuardian-Objekt geöffnete Dokument einfach geschlossen werden kann oder ob weitere Aktionen nötig sind. Im Zweifelsfall wird der Nutzer durch eine Dialogbox gefragt, wie weiter zu verfahren ist. DTConfirmClose handelt **alle denkbaren Fälle** und liefert einen der folgenden Werte zurück:

Konstante	Wert	Vorgehen
CLOSE_DISCARD	0	Die Datei soll ohne Speichern geschlossen werden, d.h. Änderungen werden verworfen.
CLOSE_SAVE	1	Die Datei soll vor dem Schließen gespeichert werden.
CLOSE_SAVE_AS	2	Die Datei ist neu oder schreibgeschützt und soll vor dem Schließen unter neuem Namen gespeichert werden.
CLOSE_CANCEL	3	Die Datei soll doch nicht geschlossen werden weil der Nutzer "Abbrechen" gewählt hat.
CLOSE_NO_FILE	4	Es ist keine Datei offen.

Entsprechend besteht die Routine BasicCloseDoc nur aus dem Aufruf der Routine DTConfirmClose sowie einer ON-SWICTH Anweisung, die die möglichen Fälle behandelt. Der Parameter saveData bestimmt, ob eventuell geänderte Daten in der Datei gespeichert werden sollen oder nicht. Ist er FALSE oder wählt der Nutzer "Änderungen vergessen" wird die Datei auf den letzten gespeicherten Stand zurückgesetzt und dann geschlossen.

```
FUNCTION  BasicCloseDoc (saveData as INTEGER) AS REAL
DIM cmd, err

  IF saveData THEN
   cmd = DTConfirmClose(ConvertObjForSDK(DocumentObj), TRUE)
  ELSE
   cmd = CLOSE_DISCARD
  END IF

  ON cmd SWITCH
  CASE CLOSE_CANCEL:        ' Abbruch
        RETURN TRUE
  CASE CLOSE_NO_FILE:       ' Keine Datei offen
        RETURN FALSE
  CASE CLOSE_DISCARD:       ' Änderungen nicht speichern
        DoRevertDoc
        END CASE
  CASE CLOSE_SAVE:          ' Änderungen speichern
        BasicSaveDoc        ' Handelt alle denkbaren Fälle
        END CASE
  CASE CLOSE_SAVE_AS:
        err = BasicSaveAsDoc()
        IF err THEN RETURN TRUE
  END SWITCH

  DocumentObj.CloseDocument
  RETURN FALSE

END FUNCTION
```

15.5 Erweiterte Dateioperationen

In diesem Abschnitt wird beschrieben, wie die Operationen "Kopieren nach", "Verschieben nach", "Umbenennen" und das Ändern der Benutzernotizen implementiert werden. Die entsprechenden Routinen **BasicCopyToDoc**, **BasicMoveToDoc**, **BasicRenameDoc** und **BasicChangeUsernotes** werden vom DocumentAndToolButtonHandler aufgerufen, wenn der Nutzer den entsprechenden Menüpunkt anklickt.

Kopieren nach ...

Die Funktion "Kopieren nach" erstellt eine Kopie des aktuellen Standes unseres Dokuments unter einem neuen Namen. Die von BasicCopyToDoc aufgerufenen Routinen haben im Folgenden dargestellten Aufgaben. Sollte ein Fehler oder ein Nutzerabbruch möglich sein wird dieser jeweils abgefragt und die Routine BasicCopyToDoc wird verlassen.

- DoEnterDocumentPath wechselt in den Pfad, in dem Dokumente normalerweise abgelegt werden.
- DTMoveCopyDialog(... FALSE) zeigt den "Kopieren nach..."-Dialog an. Die Strukturvariable ret enthält danach alle nötigen Informationen.
- DTConfirmAndDelete(ret.fileName$) prüft, ob schon eine Datei des gewünschten Namens vorhanden ist und löscht diese nach entsprechender Nachfrage beim Nutzer.
- DTCloneAndOpenFile legt eine 1:1-Kopie des aktuell geöffneten Dokuments an und öffnet diese. NewFile enthält das FILE Handle der Kopie und die globale Variable fileError enthält im (sehr unwahrscheinlichen) Falle eines Fehlers den Fehlercode.
- DoSaveDataToDoc(newFile) bringt die Kopie auf den neuesten Stand und DTCloseClone(newFile) schließt die Kopie.

Da wir an der Originaldatei nichts geändert haben brauchen wir die UI nicht upzudaten.

```
SUB BasicCopyToDoc ()
DIM ret as DialogReturnStruct
DIM err
DIM newFile AS FILE

  DoEnterDocumentPath(FALSE)

  ret = DTMoveCopyDialog(ConvertObjForSDK(DocumentObj), "", FALSE)
  IF ret.retInfo = DRI_CANCEL THEN RETURN

  err = DTConfirmAndDelete(ret.fileName$)
  IF err THEN RETURN

  newFile = DTCloneAndOpenFile(ConvertObjForSDK(DocumentObj),
                                        ret.fileName$)
  IF fileError THEN RETURN
  DoSaveDataToDoc(newFile)
  DTCloseClone(newFile)
END SUB
```

Verschieben nach ...

Die Funktion "Verschieben nach" verschiebt das aktuelle Dokument an einen neuen Ort. Dazu muss das aktuelle Dokument zunächst kopiert werden. War das erfolgreich wird es geschlossen und gelöscht. Danach wird die Kopie geöffnet. Beachten Sie, dass sich durch dieses Vorgehen das FILE Handle des Dokuments ändert.

- Die Sequenz aus DoEnterDocumentPath und DTMoveCopyDialog(... FALSE) lässt den Nutzer den neuen Namen und den neuen Pfad des Dokuments eingeben und wechselt in den neuen Pfad. DTConfirmAndDelete(ret.fileName$) stellt sicher, dass keine Datei mit dem neuen Namen am Zielort existiert.
- DTCloneFile legt die erforderliche 1:1-Kopie an. Schlägt das fehl brechen wir den Vorgang ab.

Eigentlich kann ab jetzt nichts mehr schief gehen. Trotzdem programmieren wir etwas auf Sicherheit.

- Wir merken uns den vollständigen Pfad zur aktuell geöffneten Datei sowie ihren "modified" Zustand. Beachten Sie, dass oldFile$ als String(230) deklariert ist, damit wirklich der komplette Pfad abgelegt werden kann.
- Die Methode CloseDocument schießt das aktuelle Dokument, OpenDocument (ret.FileName$) öffnet die 1:1-Kopie.
- War das Öffnen erfolgreich enthält die Instancevariable documentState einen Wert ungleich Null. In diesem Fall können wir die Originaldatei beruhigt löschen. Außerdem passen wird den Documentstatus an.
- Abschließend rufen wir DoUpdateDocButtons. Das bewirkt insbesondere, dass der Name der neuen Datei in der Titelzeile des Primary-Objekts angezeigt wird.

```
SUB BasicMoveToDoc ()
DIM ret as DialogReturnStruct
DIM err, oldState
DIM oldFile$ as STRING(230)

 DoEnterDocumentPath(FALSE)
 ret = DTMoveCopyDialog(ConvertObjForSDK(DocumentObj), "", TRUE)
 IF ret.retInfo = DRI_CANCEL THEN RETURN
 err = DTConfirmAndDelete(ret.fileName$)
 IF err THEN RETURN

 DTCloneFile(ConvertObjForSDK(DocumentObj), ret.fileName$)
 IF fileError THEN RETURN

 oldFile$ = DocumentObj.documentPath$ + "\\" + \
                   DocumentObj.documentname$
 oldState = DocumentObj.documentState AND \
                    (DOCS_MODIFIED OR DOCS_EDIT_TEMPLATE)
 DocumentObj.CloseDocument
 DocumentObj.OpenDocument(ret.FileName$)
 if ( DocumentObj.documentState ) THEN
        DocumentObj.SetDocumentState oldState, 0
        FileDelete oldFile$
 END IF
 DoUpdateDocButtons
END SUB
```

Umbenennen

Um eine Datei umbenennen zu können müssen wir sie zunächst schließen. Dann können wir sie umbenennen und mit neuem Namen wieder öffnen.

- SetCurrentPath(DocumentObj.documentPath$) wechselt in den Pfad, in dem sich das aktuell geöffnete Dokument befindet.
- Die Sequenz aus DTRenameDialog und DTConfirmAndDelete(ret.fileName$) ermöglicht dem Nutzer einen neuen Namen einzugeben und stellt sicher, dass keine Datei mit diesem Namen im aktuellen Ordner existiert.
- Wir merken uns den Namen (oldFile$) und den Zustand (oldState) des aktuell geöffneten Dokuments und schließen es dann mit CloseDocument.
- FileRename oldFile$, ret.fileName$, "m" erledigt das Umbenennen. Der Parameter "m" bewirkt, dass im (extrem unwahrscheinlichen Fall) eines Problems eine entsprechende Fehlermeldung ausgegeben wird.
- Im Fehlerfall enthält die globale Variable fileError einen Fehlerwert (ungleich Null). In diesem Fall öffnen wird die originale Datei wieder, ansonsten die umbenannte.
- In jedem Fall updaten wir den Dokumentstatus (SetDocumentState) und die UI (DoUpdateDocButtons).

```
SUB BasicRenameDoc ()
DIM ret as DialogReturnStruct
DIM err, oldState
DIM oldFile$ as STRING(32)

  SetCurrentPath(DocumentObj.documentPath$)
  ret = DTRenameDialog(ConvertObjForSDK(DocumentObj), "")
  IF ret.retInfo = DRI_CANCEL THEN RETURN
  err = DTConfirmAndDelete(ret.fileName$)
  IF err THEN RETURN

  oldFile$ = DocumentObj.documentName$
  oldState = DocumentObj.documentState AND (DOCS_MODIFIED OR
   DOCS_EDIT_TEMPLATE)
  DocumentObj.CloseDocument
  FileRename oldFile$, ret.fileName$, "m"

  IF fileError THEN
   DocumentObj.OpenDocument(oldFile$)
  ELSE
   DocumentObj.OpenDocument(ret.FileName$)
  END IF

  DocumentObj.SetDocumentState oldState, 0
  DoUpdateDocButtons

END SUB
```

Benutzernotizen ändern

Das Ändern der Benutzernotizen übernimmt die Routine DTChangeUsernotes aus der DocumentTools Library. Sie erledigt alle notwendigen Schritte, einschließlich der Anzeige der entsprechenden Dialogbox sowie der Prüfung, ob das aktuell geöffnete Dokument überhaupt Benutzernotizen unterstützt.

```
SUB BasicChangeUsernotes ()
  DTChangeUsernotes(ConvertObjForSDK(DocumentObj))
END SUB
```

15.6 Letzter Stand

Klickt der Nutzer im Menü auf den entsprechenden Button so wird im Document-AndToolButtonHandler die Routine BasicRevertDoc gestartet. Diese Routine prüft zur Sicherheit, ob das Dokument überhaupt geändert wurde und fragt dann den User, ob er sicher ist. Sodann ruft es DoRevertDoc, dass die eigentliche Arbeit erledigt und updatet dann mit DoReadDataFromDoc und DoUpdateDocButtons die UI. Wichtig ist, dass mit DoSetDocModified (FALSE) der "modified" Zustand des Dokuments zurückgesetzt wird.

```
SUB BasicRevertDoc ()
DIM ans, docState

  docState = DocumentObj.documentState
  IF (docState AND DOCS_MODIFIED) = 0 THEN RETURN

  ans = QuestionBox ("Sind Sie sicher, dass Sie alle Änderungen
                seit dem letzten Speichern verwerfen wollen?")
  IF ans <> YES THEN RETURN

  DoRevertDoc

  DoSetDocModified (FALSE)
  DoReadDataFromDoc
  DoUpdateDocButtons

END SUB
```

Nur VM-Dateien unterstützen ein echtes "Zurück zum letzten gespeicherten Stand". Wenn Sie VM-Dateien als Dokumente benutzen können Sie beliebig Daten in die Datei schreiben und trotzdem VMRevert verwenden, um den letzten gespeicherten Stand wieder herzustellen. Deshalb werden VM-Dateien von allen großen Applikationen wie GeoWrite und auch R-BASIC selbst als Dokumente benutzt.

Wenn Sie sich, wie in unserem Beispiel, gegen VM-Dateien entscheiden ist der einfachste Weg, eine "Revert"-Funktion zu unterstützen, alle Änderungen der Dokumentdaten in Instancevariablen oder globalen Variablen zu speichern und nicht in die Datei zu schreiben, bis der Nutzer explizit "Speichern" wählt. In diesem Fall besteht das Wiederherstellen des letzten gespeicherten Standes einfach darin, die (ungeänderten) Daten aus dem Dokument wieder auszulesen. DoRevertDoc hat dann nichts zu tun, weil das Auslesen der Daten von BasicRevertDoc erledigt wird, nachdem es DoRevertDoc aufgerufen hat.

Dieses Konzept hat einen großen Nachteil. Im Falle eines Fehlers gehen alle geänderten Daten verloren, weil sie nirgends in einer Datei gespeichert wurden.

Auch wenn DoRevertDoc nichts tut wird es von allen Routinen, die ein Zurücksetzen der Datei auf den letzten gespeicherten Stand erwarten, gerufen. Deshalb, wenn Sie ein Konzept haben, nicht-VM-Dateien auf ihren letzten gespeicherten Stand zurückzusetzen, so können Sie es hier implementieren.

Wenn Sie VM-Dateien benutzen braucht DoRevertDoc nur VMRevert aufzurufen.

```
SUB DoRevertDoc ()
  MsgBox "DoRevertDoc: Datei auf den letzten gespeicherten Stand
                bringen - falls  es dazu etwas zu tun gibt."
  ' Im aktuellen Konzept hat DoRevertDoc nichts zu tun
  ' Sie können die MsgBox einfach entfernen
  ' Für VM-Dateien: VMRevert(DocumentObj.documentHandle) rufen
END SUB
```

15.7 Quick Backup

Klickt der Nutzer im Menü auf den entsprechenden Button so wird im Document-AndToolButtonHandler die Routine BasicQuickBackup bzw. BasicRestoreFrom-Backup gerufen.

BasicQuickBackup legt eine Kopie der Datei im Backup-Ordner (SP_BACKUP) an. Dort werden üblicherweise keine Unterordner verwaltet. Zunächst wird ein eventuell vorhandenes älteres Backup gelöscht. DTCloneAndOpenFile legt eine 1:1 Kopie der aktuell offenen Datei an und öffnet diese. Im Fehlerfall gibt es eine Fehlermeldung, ansonsten wird die Backupkopie mit DoSaveDataToDoc auf den neuesten Stand gebracht und danach geschlossen. In jedem Fall rufen wir DoUpdateDocButtons um den "Aus Backup wiederherstellen"-Button zu updaten.

```
SUB BasicQuickBackup ()
DIM fileName$ ', docState, ro
DIM docPath$ as string(200)
DIM backupFile as FILE

   fileName$ = DocumentObj.documentName$
   SetStandardPath SP_BACKUP

   FileDelete fileName$
   backupFile = DTCloneAndOpenFile(ConvertObjForSDK(DocumentObj),
                                        fileName$)
   IF fileError THEN
        MsgBox "Konnte Backup-Datei nicht anlegen. Fehlercode: "
                                        + ErrorText$(fileError)
   ELSE
        DoSaveDataToDoc(backupFile)
        DTCloseClone(backupFile)
   End IF

   DoUpdateDocButtons

END SUB
```

BasicRestoreFromBackup stellt eine Datei aus einer Backupkopie wieder her. Dazu gehen wir folgendermaßen vor:

- Wir merken uns Name (fileName$) und Pfad (docPath$) der aktuell offenen Datei.
- Nach dem Wechsel in den Backup-Ordner prüfen wir mit der Routine DTCheck-FileType aus der DocumentTools Library ob eine Backupdatei existiert und ob diese kompatibel zur aktuell offenen Datei ist. Das umfasst den Dateityp, den Dateinamen und für GEOS bzw. VM-Dateien auch das Token bzw. das CreatorToken. Im Fehlerfall brechen wir den Prozess ab.
- Das "Wiederherstellen" der Datei besteht aus vier Schritten:
  1. Schließen des aktuellen Dokuments.
  2. Kopieren der Backupkopie an die Stelle des aktuellen Dokuments. Das alte Dokument wird dabei automatisch überschrieben.
  3. Öffnen der herkopierten Backupkopie.
  4. Update der UI mit DoReadDataFromDoc und DoUpdateDocButtons.

```
SUB BasicRestoreFromBackup ()
DIM docPath$ as String(200)
DIM fileName$, err

   fileName$ = DocumentObj.documentname$
   docPath$ = DocumentObj.documentPath$

   SetStandardPath SP_BACKUP
   err = DTCheckFileType(ConvertObjForSDK(DocumentObj),
                                          fileName$, "*")
   IF err THEN
        MsgBox "Kann Backup nicht wieder herstellen. Fehlercode:
                                          " + ErrorText$(err)
        RETURN
   End IF

   DocumentObj.CloseDocument
   FileCopy fileName$, docPath$ + "\\" + fileName$
   SetCurrentPath docPath$
   DocumentObj.OpenDocument fileName$

   DoReadDataFromDoc
   DoUpdateDocButtons

END SUB
```

15.8 Verwendung von Muster-Dateien

Muster-Dateien werden immer in einem Unterordner des Ordners "USERDATA\TEMPLATE" gespeichert. Klickt der Nutzer im Menü auf einen der zugehörigen Buttons so wird im DocumentAndToolButtonHandler eine der Routinen BasicOpenTemplate oder BasicSaveAsTemplate gerufen.

Die Routine BasicOpenTemplate öffnet eine Musterdatei indem entweder eine neue Datei angelegt wird oder das Muster zum Bearbeiten geöffnet wird.

- DTOpenTemplateDialog erlaubt es dem Nutzer eine Musterdatei auszuwählen. Standardmäßig ist die Option "Zum Bearbeiten" deaktiviert und ret.retInfo enthält den Wert DRI_READ_ONLY. Aktiviert der Nutzer die genannte Option enthält ret.retInfo den Wert DRI_OK.
  srcFile$ speichert den kompletten Pfad zu ausgewählten Musterdatei.
- BasicCloseDoc(TRUE) schließt die aktuell geöffnete Datei und fragt den Nutzer gegebenenfalls ob er seine Änderungen speichern will usw. Wählt der Nutzer "Abbrechen" wird die Datei nicht geschlossen und wir verlassen die Routine.
- Für den Fall, dass der Nutzer die Musterdatei bearbeiten will (ret.retInfo = DRI_OK) öffnen wir die Musterdatei selbst (ret.fileName$) und teilen dem DocumentGuardian-Objekt mit, dass wir eine Musterdatei bearbeiten (DocumentObj.SetDocumentState DOCS_EDIT_TEMPLATE, 0). Dieses Flag

bewirkt nur, dass der nächste "Öffnen"-Dialog den Dokument-Ordner anzeigt, und nicht dem der aktuell offenen Datei (den Template-Ordner).

- Wenn der Nutzer das Muster verwenden will um eine neue Datei anzulegen gehen wir folgendermaßen vor:
  1. DoEnterDocumentPath(TRUE) wechselt in den Pfad, wo neue Dokumente abgelegt werden.
  2. DTFindNameForNew findet einen passenden (und noch unbenutzten) Namen für das neue Dokument.
  3. FileCopy srcFile$, fileName$, "m" kopiert das Musterdokument unter dem neuen Namen in den aktuellen Ordner (siehe Punkt 1.)
  4. Für den Fall dass alles gut gegangen ist öffnen wir das neue Dokument und markieren es als "unbenannt".
- Abschließend passen wir die UI mit der Standardsequenz aus DoReadData-FromDoc und DoUpdateDocButtons an.

```
SUB BasicOpenTemplate ()
DIM fileName$ as string(32)
DIM srcFile$ as String(240)
DIM err
DIM ret as DialogReturnStruct

   ret = DTOpenTemplateDialog(ConvertObjForSDK(DocumentObj), "")
   IF ret.retInfo = DRI_CANCEL THEN RETURN
   srcFile$ = currentPath$ + "\\" + ret.fileName$

   err = BasicCloseDoc(TRUE)
   IF err THEN RETURN

   IF ret.retInfo = DRI_OK THEN          ' => zum Bearbeiten
       DocumentObj.OpenDocument ret.fileName$
       DocumentObj.SetDocumentState DOCS_EDIT_TEMPLATE, 0
   ELSE
       DoEnterDocumentPath(TRUE)
       fileName$ = DTFindNameForNew(ConvertObjForSDK(DocumentObj))
       FileCopy srcFile$, fileName$, "m"
       IF fileError = 0 THEN
             DocumentObj.OpenDocument fileName$
             DocumentObj.SetDocumentState DOCS_UNTITLED, 0
       End IF
   End IF

   DoReadDataFromDoc
   DoUpdateDocButtons

END SUB
```

BasicSaveAsTemplate speichert eine Datei als Muster im Template-Ordner. Dazu ruft es zuerst DTSaveAsTemplateDialog. Diese Routine erlaubt es dem Nutzer einen Namen und ggf. einen anderen Pfad für die Musterdatei einzugeben. Die

nächsten Schritte sind identisch mit dem "normalen" Speichern einer Datei mit einem neuen Namen. Deswegen können wir diese Aufgabe an InternalSaveAs delegieren. Diese Routine ist im Abschnitt 15.4.4 (Speichern unter neuem Namen) beschrieben. Sie speichert die aktuell offene Datei mit einem neuen Namen und öffnet diese zum Bearbeiten.

Nun informieren wir das DocumentGuardian-Objekt, dass eine Musterdatei in Bearbeitung ist und wir haben die Möglichkeit "Muster-typische" Änderungen an der Datei vorzunehmen. Ob es da etwas gibt und was das ist hängt wieder vom Ihrem Programm ab. Abschließend geben wir noch eine Erfolgsmeldung an den Nutzer aus.

```
SUB BasicSaveAsTemplate ()
DIM err
DIM ret as DialogReturnStruct

   ret = DTSaveAsTemplateDialog(ConvertObjForSDK(DocumentObj),
                                "", FALSE)
   IF ret.retInfo = DRI_CANCEL THEN RETURN

   err = InternalSaveAs(ret)
   IF err THEN RETURN

   DocumentObj.SetDocumentState DOCS_EDIT_TEMPLATE, 0
   MsgBox("BasicSaveAsTemplate: Datei gespeichert. Hier eventuell
                  Sonderaufgaben für \"Muster\" erledigen")
   MsgBox "Die Datei wurde als Muster gespeichert und zum
                                Bearbeiten geöffnet."
   RETURN

END SUB
```

Das GEOS-System hat für "Muster" Dateien ein spezielles Attribut, das dem Dokument-Interface mitteilt, dass es sich nicht um ein normales Dokument handelt. R-BASIC unterstützt dieses Attribut nicht. Wenn Sie möchten können Sie an dieser Stelle einen bestimmten Wert in Ihrem Muster-Dokument setzen, der das Dokument als "Muster" kennzeichnet. Diesen können Sie beim Öffnen des Dokuments auslesen und entsprechend (analog zu BasicOpenTemplate) reagieren.

15.9 Schnittstelle zum GEOS Dateisystem

Ein Programm, das mit Dokumenten arbeitet, muss mit folgenden Situationen umgehen können:

- Der Nutzer öffnet im GeoManager ein zu dem Programm gehörendes Dokument. In diesem Fall unterscheidet das System zwei Fälle. Wenn das Programm noch nicht läuft wird es gestartet und der OnStartup-Handler (in unserem Fall die Routine **DocStartupHandler**) des Programms wird ausgeführt. Läuft das Programm bereits wird stattdessen der OnConnection-Handler des Programms ausgeführt (in unserem Fall die Routine **DocConnection-Handler**). In beiden Fällen wird den Handlern der komplette Pfad zur Datei übergeben und das Programm muss in der Lage sein diese Datei zu öffnen. Das erledigen wir mit der Routine **OpenExternalFile**.
- Der Nutzer selektiert im GeoManager ein Dokument und wählt den Menüpunkt "Drucken". Diese Situation wird nicht hier, sondern im Objekthandbuch, beim PrintControl-Objekt, Kapitel 4.14.8, besprochen.
- Der Nutzer schließt das Programm während noch ein Dokument offen ist. Dann muss der OnExit-Handler (in unserem Fall die Routine **DocExitHandler**) dafür sorgen, dass die Dokumentendatei geschlossen wird. Falls erforderlich muss der Nutzer vorher gefragt werden, ob er Änderungen in der Datei speichern will.
- GEOS fährt bei offenem Programm herunter. Auch hier ist der OnExit-Handler gefragt. Er muss dafür sorgen, dass die Datei so geschlossen wird, dass sie beim Wiederhochfahren automatisch geöffnet werden kann. Diese Aufgabe delegieren wir an das DocumentGuardian-Objekt. Vorher müssen wir eventuell in globalen Variablen gecachte Daten sichern.
- Wenn GEOS wieder hochfährt muss die beim Herunterfahren geschlossene Datei automatisch wieder geöffnet werden. Auch das delegieren wir komplett an das DocumentGuardian-Objekt. Danach müssen wir gegebenenfalls globale Variablen wiederherstellen.

Alle drei Handler müssen wie folgt im UI-Code als Handler des Application-Objekts vereinbart werden. Wenn Sie bereits einen entsprechenden Handler haben reicht es, wenn Sie den Code in den bereits definierten Handler verschieben.

```
Application MyAppObject
  OnStartup = DocStartupHandler
  OnExit = DocExitHandler
  OnConnection = DocConnectionHandler
  <..>
End Object
```

Außerdem benutzen wir die Tatsache, dass alle Instancevariablen von Generic-Class Objekten eine Systemrestart automatisch überleben. Das Document-Guardian-Objekt merkt sich also z.B. den Namen und den Pfad der aktuell offenen Datei automatisch, ohne unser Zutun. Wenn wir weiterhin darauf achten, dass alle geänderten Daten des Dokument in irgendwelchen Instancevariablen gespeichert sind (das ist z.B. automatisch der Fall beim Text eines Memo-Objekts, dem ausgewählten Eintrag einer Liste oder der aktuellen Farbe bei einem ColorSelector), stehen uns diese Daten nach einem Systemneustart automatisch wieder zur Verfügung. Kümmern müssen Sie sich nur, wenn Sie Daten in globalen

Variablen haben. Diese könnten Sie z.B. in der Instancevariablen "document-UserData" des DocumentGuardian-Objekts "retten".

Um eine DOS-Datei mit einem Programm zu verknüpfen muss in der GEOS.INI in der Kategorie [filemanager] unter "fileNameTokens" ein Eintrag der Form

*.EXT="DTOK", 0, "AppT", 16600

existieren, wobei *.EXT die DOS-Datei beschreibt, "DTOK", 0 das Token ist, das der GeoManager zur Anzeige der Datei verwenden soll und "AppT", 16600 das Token unseres Programms ist.
R-BASIC unterstützt das Setzen eines solchen Eintrags nicht. Der Nutzer muss das selbst tun, z.B. mit Hilfe des Voreinstellungs-Moduls.

Für GEOS- und VM-Dateien müssen wir nicht in die GEOS.INI eingreifen. Wir müssen der Datei nur ein CreatorToken zuweisen. Damit weiß das System zu welchem Programm die Datei gehört. Sinnvoller Weise geben wir der Datei auch noch ein eigenes Token. Das alles erledigt das DocumentGuardian-Objekt für uns. Wir müssen nur Token und CreatorToken bei der Konfiguration des Objekts angeben (siehe Kapitel 15.3, Routine DoInitDocumentGuardian).

Der OnStartup-Handler

Wie oben beschrieben muss der OnStartup-Handler (er heißt DocStartupHandler) unterscheiden, ob das Programm neu startet oder ob GEOS gerade wieder hochfährt. Außerdem muss er wissen, ob eine Datendatei (Dokument) übergeben wurde oder nicht. Die entsprechende Information ist im Parameter "flags" zu finden. Ist das Bit AF_RESTORE gesetzt fährt GEOS nach einem Shutdown wieder hoch. Das Bit AF_DATA_FILE ist gesetzt, wenn eine Datei an den Handler übergeben wurde. Das ermöglicht uns folgendes Vorgehen:

- Wir prüfen zunächst das Bit AF_RESTORE. Ist es gesetzt rufen wir die Methode HandleRestart des DocumentGuardian-Objekts. Diese erledigt die notwendigen Schritte. DoReadCachedData stellt bei Bedarf die globalen Variablen wieder her.
- Ist das Bit AF_RESTORE nicht gesetzt startet das Programm gerade neu. In diesem Fall müssen wir unbedingt das DocumentGuardian-Objekt initialisieren. Das erledigt die weiter oben beschriebene Routine DoInitDocumentGuardian.
- Falls eine Datendatei übergeben wurde (das Bit AF_DATA_FILE ist gesetzt) öffnen wir diese mit OpenExternalFile, andernfalls blenden wir mit DTShow-NewOpenDialog den "Neu/Öffnen" Dialog ein. DTShowNewOpenDialog wartet nicht bis der Nutzer etwas eingibt sondern öffnet nur den Dialog und kehrt dann zurück, so dass die Startup-Sequenz des Programms fortgesetzt werden kann. Der Nutzer kann irgendwann später einen Button im "Neu/Öffnen" Dialog anklicken und die DocumentTools Library ruft dann, wie vorn beschrieben, den DocumentAndToolButtonHandler.

```
SYSTEMACTION DocStartupHandler

  IF flags AND AF_RESTORE THEN
    DocumentObj.HandleRestart
    DoReadCachedData (DocumentObj)
    RETURN
  End IF

  DoInitDocumentGuardian(DocumentObj)

  IF flags AND AF_DATA_FILE THEN
    OpenExternalFile(dataFile$)
  ELSE
    DTShowNewOpenDialog(ConvertObjForSDK(DocumentObj),
        NOF_STARTUP + NOF_NEW_OPEN_TEMPLATE + NOF_CONFIG +
        NOF_IMPORT, "")
  End IF

END ACTION
```

Der OnExit-Handler

Der OnExit-Handler namens DocExitHandler hat nicht zu tun, wenn gar kein Dokument offen ist. Das wird deswegen zuerst abgefragt.
Ist ein Dokument offen muss er zwischen zwei Fällen unterscheiden:

- GEOS fährt herunter. In diesem Fall ist das Bit AF_SHUTDOWN im Parameter "flags" gesetzt. Dann sichern wir die wichtigen globalen Variablen (mit DoSaveCachedData) und rufen die Methode HandleShutdown des Document-Guardian-Objekts. Diese erledigt die notwendigen Schritte. Dazu gehört vor allem die Datei zu schließen, sich aber den Namen und den Pfad zu merken, so dass die Methode HandleRestart genau diese Datei wieder öffnen kann.
- Das Programm schließt. In diesem Fall muss das Dokument geschlossen werden. Vorher müssen wir den Nutzer fragen, ob er eventuelle Änderungen speichern will. Dazu verwenden wir die im Kapitel 15.4.5 (Schließen des Dokuments) beschriebene Funktion DTConfirmClose. Der Parameter "FALSE" bewirkt, dass der Nutzer die Option "Abbrechen" nicht hat. Je nach zurückgegebenem Wert rufen wir BasicSaveDoc (CLOSE_SAVE), BasicSaveAsDoc (CLOSE_SAVE_AS) oder nichts davon (CLOSE_DISCARD, Änderungen verwerfen).
  Abschließend können wir mit der Methode CloseDocument die Datei schließen.

```
SYSTEMACTION DocExitHandler
DIM cmd

   IF DocumentObj.documentHandle = NullFile() THEN RETURN
   IF flags AND AF_SHUTDOWN THEN
        DoSaveCachedData (DocumentObj)
        DocumentObj.HandleShutdown
        RETURN
   End IF
```

```
   cmd = DTConfirmClose(ConvertObjForSDK(DocumentObj), FALSE)
   ON cmd SWITCH
   CASE CLOSE_SAVE:            ' Änderungen speichern
        BasicSaveDoc           ' Handelt alle denkbaren Fälle
        END CASE
   CASE CLOSE_SAVE_AS:
        BasicSaveAsDoc()
   END SWITCH

   DocumentObj.CloseDocument

END ACTION
```

Der OnConnection-Handler

Der OnConnection-Handler (er heißt DocConnectionHandler) wird gerufen, wenn der Nutzer ein Dokument im GeoManager doppelklickt, das zugehörige Programm aber schon läuft. Der vollständige Pfad zu diesem Dokument wird dem Handler im Parameter dataFile$ übergeben. Da es nicht auszuschließen ist, dass GEOS den Handler auch in anderen Zusammenhängen ruft, fragen wir das Bit AF_DATA_FILE ab, bevor wir OpenExternalFile zum Öffnen der Datei rufen.

```
SYSTEMACTION DocConnectionHandler

  IF flags AND AF_DATA_FILE THEN
  OpenExternalFile(dataFile$)
  End IF

END ACTION
```

OpenExternalFile

Die Routine OpenExternalFile erledigt alles was nötig ist um eine Datei zu öffnen, die einem der Handler DocStartupHandler oder DocConnectionHandler übergeben wurde.

```
SUB OpenExternalFile (file$ as string(235))
DIM err, n, state
DIm fileName$ as String(32)
DIM path$ as String(235)

  state = DocumentObj.documentState
  IF state THEN
    err = BasicCloseDoc(True)
    IF err THEN RETURN
  END IF
```

```
  path$ = file$
  n = InStr("\\", path$)
  WHILE n <> 0
    path$ = Right$(path$, len(path$) - n)
    n = InStr("\\", path$)
  WEND

  fileName$ = path$
  path$ = left$(file$, len(file$) - len(fileName$) - 1)

  SetCurrentPath path$
  DocumentObj.OpenDocument fileName$

  DoReadDataFromDoc
  DoUpdateDocButtons

END SUB
```

Zunächst prüfen wir ob noch eine Datei offen ist (die Variable state ist dann ungleich Null). In diesem Fall erledigt BasicCloseDoc das Schließen der Datei mit vorheriger Nachfrage beim Nutzer. Entscheidet sich der Nutzer die Datei doch nicht zu schließen liefert BasicCloseDoc TRUE und wir verlassen die Routine OpenExternalFile.

Der nächste Schritt ist das Separieren von Pfad und Dateinamen. Dafür verwenden wir die lokale Variable path$ denn der Parameter file$ wird später noch gebraucht. Die WHILE Schleife sucht jeweils den nächsten Backslash. Aus "C:\GEOS\DOCUMENT\NAME.EXT" wird so schrittweise "GEOS\DOCUMENT\NAME.EXT", "DOCUMENT\NAME.EXT" und schließlich "NAME.EXT". Das ist der Dateiname, also speichern wir ihn in fileName$. Der Pfad ist dann alles links davon, mit Ausnahme des letzten Backslash-Zeichens.

Jetzt können wir mit SetCurrentPath in den richtigen Ordner wechseln und mit der Methode OpenDocument das Dokument öffnen. Abschließend rufen wir DoReadDataFromDoc und DoUpdateDocButtons um die UI zu updaten.

15.10 Ein einfaches Beispiel

Am Beispiel der Programms "Yellow Notes", das im Ordner "Beispiele\Objekte \Dateiarbeit" gefunden werden kann, soll gezeigt werden, wie man die zum Dokumentinterface gehörende Routinen an das eigene Programm anpassen kann.

Kernobjekte des Programms sind ein Textobjekt (YNotesText) und ein Menu (YNotesColorMenu) mit zwei ColorSelektoren (YNotesTextColor und YNotesBack-Color) für die Vordergrund- und die Hintergrundfarbe.

Sehr häufig ist es sinnvoll, die Dokumentdaten in einer Struktur zu speichern. Das vereinfacht den Zugriff auf die Daten und deren Verwaltung, insbesondere das Speichern in einer Dokumentdatei, enorm. Für das "Yellow-Notes"-Beispiel benötigen wir die Farben von Text und Hintergrund sowie den Notiztext selbst. Außerdem haben wir 8 Word Reserve vorgesehen, die wir später zur kompatiblen Erweiterung des Programms verwenden können.

```
STRUCT NotesData
  backcolor, textColor as word
  reserve[8] as word
  text as String(1024)
End STRUCT
```

Als Dokumentdatei wählen wir eine GEOS Datendatei. Eine DOS-Datei sollte man nur verwenden, wenn es erforderlich ist. Der Zugriff auf eine GEOS Datendatei ist genau so einfach wie der auf eine DOS-Datei, aber man kann ein Token und ein CreatorToken vergeben, so dass die Verknüpfung mit dem zugehörigen Programm ohne Eingriff in die GEOS.INI erfolgt.

Wie am Anfang des Kapitels beschrieben beschränkt sich die Anpassung des Dokument-Interfaces auf die folgenden Routinen:

- DoInitDocumentGuardian
- BasicCreateNewDoc
- DoUpdateDocButtons
- DoReadDataFromDoc
- DoSaveDataToDoc
- DoEnterDocumentPath
- DoSetDocModified
- DoRevertDoc

In der Routine **DoInitDocumentGuardian** müssen alle das Dokument betreffenden Daten an das eigene Programm angepasst werden. Wichtig ist, dass der Wert für CreatorToken dem AppToken-Statement im UI-Code des Application-Objekts entspricht.

```
SUB DoInitDocumentGuardian(guardian as object)
DIM dc as DocumentConfigStruct

  guardian.buttonhandler = DocumentAndToolButtonHandler

  dc.noDocumentString$ = "leer"
  dc.nameForNew$ = "Notiz "

  dc.fileType = GFT_DATA
  dc.creatorToken.tokenChars = "YNot"
  dc.creatorToken.manufid = 16600
  dc.token.tokenChars = "YNOD"
  dc.token.manufid = 16600

  dc.matchFlags = DOC_MATCH_TOKEN
  guardian.ConfigData = dc

END SUB
```

In der Routine **BasicCreateNewDoc** muss der Teil angepasst werden, der für das Initialisieren des neu angelegten Dokuments zuständig ist. Im Yellow Notes Beispiel wird dazu die Sub YNotesInitializeDocument aufgerufen. Sie belegt eine NotesData-Struktur mit den Standardfarben und einem leeren Text und schreibt sie dann in die neu angelegt Datei. Token und CreatorToken werden automatisch vom DocumentGuardian-Objekt gesetzt.

```
SUB YNotesInitializeDocument ()
DIM notes as NotesData

  notes.textColor = BLACK
  notes.backColor = YELLOW
  notes.text = ""

  FileSetPos DocumentObj.documentHandle , 0
  FileWrite DocumentObj.documentHandle , notes, sizeof(NotesData)

END SUB
```

Die Routine **DoUpdateDocButtons** hat die Aufgabe, die UI des Programms zu enablen oder zu disablen, je nachdem ob ein Dokument offen ist und welchen Status es hat. Im Yellow Notes Beispiel ruft sie dazu die SUB YNotesUpdateUI, die im Folgenden gezeigt ist. DocumentObj.documentState ist ungleich Null, wenn ein Dokument offen ist. Dann wird die UI enabled, ansonsten wird sie disabled.

```
SUB YNotesUpdateUI ()
  IF DocumentObj.documentState THEN
    YNotesText.enabled = TRUE
    YNotesColorMenu.enabled = TRUE
  ELSE
    YNotesText.enabled = FALSE
    YNotesColorMenu.enabled = FALSE
  End IF
END SUB
```

Die Routine **DoReadDataFromDoc** wird jedes Mal gerufen, wenn Daten aus der Datei gelesen werden sollen oder ein Dokument geschlossen wurde. Sie liest die NotesData-Struktur aus der Datei und verteilt die Informationen an die entsprechenden UI-Objekte. Falls keine Datei offen ist muss sie dafür sorgen, dass das Objekt YNotesText leer ist. Das Enablen bzw. Disablen der UI-Objekte übernimmt die Routine DoUpdateDocButtons.

```
SUB DoReadDataFromDoc ()
DIM notes as NotesData

  IF DocumentObj.documentHandle == NullFile() THEn
    YNotesText.text$ = ""
  ELSE
    ' Text und Farben updaten
    FileSetPos DocumentObj.documentHandle, 0
    notes = FileRead DocumentObj.documentHandle,sizeof(NotesData)
    YNotesText.textColor = notes.textColor
    YNotesText.backColor = notes.backColor
    YNotesText.text$ = notes.text
    YNotesTextColor.csColor = notes.textColor
    YNotesBackColor.csColor = notes.backColor
  End IF

END SUB
```

Die Routine **DoSaveDataToDoc** schreibt die aktuellen Daten in die Datei, indem sie die Struktur "notes" mit den aktuellen Werten, gelesen vom Textobjekt und den ColorSelektoren, belegt und sie dann mit einem einzigen FileWrite in die Datei schreibt. Wir dürfen natürlich nicht vergessen vorher mit FileSetPos die korrekte Schreibposition anzuwählen.

```
SUB DoSaveDataToDoc (fh as FILE)
DIM notes AS NotesData
  notes.textColor = YNotesText.textColor
  notes.backColor = YNotesText.backColor
  notes.text = YNotesText.text$
  FileSetPos fh, 0
  FileWrite fh, notes, sizeof(NotesData)
END SUB
```

DoEnterDocumentPath wechselt in dem Pfad, in dem die Dateien angelegt werden sollen. Wir entscheiden uns bei neuen Dateien für den GEOS-Top-Ordner und für die Notiz-Dateien selbst für den Document-Ordner ohne Unterordner.

```
SUB DoEnterDocumentPath (forNew as Integer)
   IF forNew THEN
    SetStandardPath SP_TOP
   ELSE
    SetStandardPath SP_DOCUMENT
   END IF
END SUB
```

In der Routine **DoSetDocModified** gibt es eine wichtige Anpassung: Wir müssen sicherstellen, dass nach dem Speichern des Dokuments (in diesem Fall wird DoSetDocModified automatisch mit dem Parameter FALSE aufgerufen) die nächste Nutzereingabe wieder den OnModified Handler des Textobjekts ruft. Dazu setzen wir den modified-Status des Textobjekts zurück.

```
SUB DoSetDocModified (modi as INTEGER)

  IF  modi THEN
    ' ist schon "modified"? => Return
    IF DocumentObj.documentState AND DOCS_MODIFIED THEN RETURN
    DocumentObj.SetDocumentState DOCS_MODIFIED, 0
  ELSE
    ' ist schon "not modified"? => Return
    IF (DocumentObj.documentState AND DOCS_MODIFIED) = 0 \
                                                    THEN RETURN
    DocumentObj.SetDocumentState 0, DOCS_MODIFIED
    YNotesText.modified = FALSE
  End IF

  DoUpdateDocButtons

END SUB
```

Da sich das Zurücksetzen des Dokuments auf den letzten gespeicherten Stand in unserem einfachen Konzept darauf beschränkt, die die Dokument-Daten aus der (ungeänderten) Datei wieder auszulesen, hat die Routine **DoRevertDoc** nichts zu tun. Sie könnte komplett aus dem Code entfernt werden.

```
SUB DoRevertDoc ()
END SUB
```

Zusätzlich benötigt das Yellow Notes Beispiel ein paar weitere Routinen. Die wichtigsten davon sind der OnModified-Handler des YNotesText-Objekts und der

ColorChangedHander der beiden ColorSelektoren, da diese das Dokument als "modified" markieren müssen, wenn der Nutzer etwas ändert.

Der OnModified Handler des Textobjekts ist sehr einfach. Er ruft nur DoSetDocModified (TRUE). Diese Routine informiert das DocumentGuardian-Objekt und ruft DoUpdateDocButtons. Mehr ist nicht zu tun.

```
TEXTACTION TextModifedHandler
  DoSetDocModified (TRUE)
END ACTION
```

Beide ColorSelektoren haben den gleichen ColorChangedHander. Seine Aufgabe ist es, dem Text-Objekt eine neue Vorder- und Hintergrundfarbe zuzuweisen sowie das Dokument als "geändert" zu markieren.
ColorSelector Objekte haben die Eigenart, dass der Handler öfter gerufen wird, als es für unsere Zwecke sinnvoll ist, z.B. wenn sie erstmalig auf dem Schirm erscheinen. Das könnte dazu führen, dass das Dokument als "geändert" markiert wird, obwohl es eigentlich nicht geändert wurde. Deswegen fragen wir die aktuellen Farben des Textobjekts ab und rufen DoSetDocModified (TRUE) nur dann, wenn sie sich wirklich geändert haben.

```
COLORACTION NewColorHandler
dim tc, bc

  tc = YNotesText.textColor
  bc = YNotesText.backColor

  YNotesText.textColor = YNotesTextColor.csIndexColor
  YNotesText.backColor = YNotesBackColor.csIndexColor

  IF (tc <> YNotesText.textColor) \
    OR (bc <> YNotesText.backColor) THEN
    DoSetDocModified (TRUE)
  End IF
END ACTION
```

Den kompletten Quellcode für dieses Beispiel sowie die Iconeditor-Datei mit den Iconbildern findet man im Ordner "Beispiele\Objekte\Dateiarbeit".

16 Timer

Timer erlauben es, einen Actionhandler in bestimmten Zeitabständen automatisch aufzurufen. Man kann sich das so vorstellen, als ob jemand einen Button in regelmäßigen Abständen drückt. Damit kann man beispielsweise eine blinkende Schrift realisieren, eine Spielfigur über das Spielfeld bewegen oder eine Uhr weiterzählen.

Die Routine TimerStart aktiviert einen Timer. Sie erwartet den Namen des Actionhandlers, der aufgerufen werden soll, sowie einen oder zwei numerische Werte. Der erste Wert gibt an, wie lange es dauern soll, bis der Timer das erste Mal auslöst. Die Zeitangabe erfolgt in "tics", das sind 1/60s. Der zweite Wert gibt das Zeitintervall an, in dem der Timer danach periodisch auslösen soll (ebenfalls in tics). Wird der zweite Wert nicht angegeben oder ist er Null, so löst der Timer nur genau einmal aus (Single-Shot Timer). Das maximale Zeitintervall beträgt jeweils 65535 tics, das entspricht etwa 18 Minuten.
Um einen Timer zu stoppen verwenden Sie TimerStop. Single-Shot Timer brauchen nicht gestoppt zu werden. Es ist ein guter Stil alle Timer am Programmende, vorzugsweise im OnExit-Handler, zu stoppen. Sollten Sie das vergessen, stoppt das System die aktiven Timer.

Wenn ein Timer auslöst erzeugt er ein BASIC-Event, das wie allen anderen Events (Aktivieren eines Button, Klick in eine Liste usw.) behandelt wird. Das bedeutet:

- Timerevents unterbrechen laufende Actionhandler nicht. Das Timerevent wird erst behandelt, wenn der laufende Actionhandler beendet ist.
- Timerevents haben keine erhöhte Priorität. Sie reihen sich wie jedes andere Ereignis in der Ereigniswarteschlage hinten ein.

Bei sehr schnellen Timern kann es vorkommen, dass der Timer bereits wieder auslöst, bevor das letzte Timerevent behandelt wurde. Damit diese Situation nicht zu einem Überlaufen der Warteschlange führt, stellt der Eventmanager sicher, dass sich für jeden Timer maximal ein Event in der Warteschlage befindet. Zu schnell aufeinander folgende Events werden verworfen.

Verwechseln Sie Timerevents nicht mit den Befehlen Delay und Pause! Delay und Pause unterbrechen die Abarbeitung eines Handlers für eine bestimmte Zeit während Timer einen eigenen Handler aufrufen.

TimerStart

TimerStart aktiviert einen Timer. Das kann eine einmaliger Timer (Single-Shot-Timer) oder ein periodischer Timer sein.

Syntax: **<th>** = **TimerStart** (**<Handler>**, **tics1** [, **tics2**])
<th>: Variable vom Typ **HANDLE**.
Der Wert wird für TimerStop benötigt.
<Handler>: Name des ActionHandlers, der vom Timer aufgerufen werden soll
Er muss als TimerAction deklariert sein.
tics1: Zeit, bis der Timer erstmalig auslöst (in 1/60s)
tics2: Intervall (in 1/60s), in dem der Timer periodisch auslösen soll. Wird tics2 nicht angegeben (oder wenn er Null ist) löst der Timer nur einmal aus (Single-Shot-Timer).
Erlaubte Werte für tics1 und tics2: 0 ... 65535

TimerStop

TimerStop hält einen Timer an. Es erwartet das Handle, das von TimerStart zurückgegeben wurde. Es ist explizit erlaubt:
• Einen Timer zu stoppen, der bereits gestoppt wurde.
• Einen Single-Shot-Timer zu stoppen, der bereits ausgelöst hat.

Hinweis: TimerStop entfernt keine Timerevents aus der Warteschlange. Falls sich beim Aufruf von TimerStop noch ein Timerevent in der Warteschlange befindet, so wird dieses noch ausgeführt.

Syntax: **TimerStop <th>**
<th>: Handle, das von TimerStart geliefert wurde

TimerAction

Actionhandler, die von einem Timer aufgerufen werden, müssen als TimerAction deklariert sein.

Handler-Typ	Parameter
TimerAction	(sender as object, actionData as integer)

Der Parameter "sender" enthält das Application-Objekt des Programms, der Parameter "actionData" ist unbenutzt und enthält den Wert Null.

Beispiele. Den folgenden Code finden Sie komplette im R-BASIC Beispiel "Datum und Zeit\TimerDemo".

Beispiel 1: Blinkende Schrift

Eine globale Variable z bestimmt, ob die Schrift gezeigt wird oder nicht. Das TimerHandle th ist ebenfalls global, damit wir dem Timer wieder anhalten können.

```
DIM z
DIM th as HANDLE
```

Mit TimerStart aktivieren wir die blinkende Schrift. Der erste Timerevent soll sofort ausgelöst werden (zweiter Parameter ist Null), dann soll der Timer alle 0,5 Sekunden auslösen (dritter Parameter: 30 tics).

```
th = TimerStart (TimerBlink, 0, 30)
```

TimerStop schaltet die blinkende Schrift wieder aus. Wir kümmern uns nicht darum, ob die Schrift gerade zu sehen ist oder nicht.

```
TimerStop th
```

Der eigentliche TimerHandler prüft die globale Variable z. Ist sie ungleich Null wird dein Leertext ausgegeben und z auf Null gesetzt. Beim nächsten Handleraufruf ist z dann Null und der Text selbst wird ausgegeben. z wird auf 1 gesetzt.

```
TIMERACTION TimerBlink
  IF z THEN
    Print at 3, 5; "                    "
    z = 0;
  ELSE
    print at 3, 5;"R-BASIC Timer Demo"
    z = 1
  End IF

END ACTION
```

Beispiel 2: Willkommensbox

Viele Programme zeigen am Start eine Infobox an, die dann von allein wieder verschwindet. Dafür eignet sich ein Single-Shot-Timer. Im OnStartup-Handler öffnen wir die Dialogbox und starten den Timer. Das Timerhandle th2 wird bei Single-Shot-Timern nicht weiter gebraucht.

```
SYSTEMACTION DemoStartupHandler
  DemoStartupDialog.Open
  th2 = TimerStart ( StartupTimerHandler, 180) ' 3 sek.
END ACTION
```

Der Timerhandler muss nur die Dialogbox schließen.

```
TIMERACTION StartupTimerHandler
  DemoStartupDialog.Close
END ACTION
```

Die Dialogbox selbst sollte das Attribut DA_HIDDEN_UNTIL_OPENED gesetzt haben, damit das System keinen Button erzeugt, mit dem man die Dialogbox manuell öffnen kann. Die Anweisung "modal = APP_MODAL" ist auskommentiert. Sie würde bewirken, dass der Nutzer nicht mit dem Programm interagieren kann, solange die Dialogbox noch offen ist.

```
Dialog DemoStartupDialog
  Caption$ = "Willkommen!"
  Children = TimerStartupText
  attrs = DA_HIDDEN_UNTIL_OPENED
  'modal = APP_MODAL
End OBJECT
```

Tipps & Tricks

- Actionhandler vom Typ TimerAction sind kompatibel mit dem Typ ButtonAction. Das heißt, Sie können einem Button einen Timer-Handler als ActionHandler zuweisen und so ihren Timerhandler komfortabel testen. In diesem Fall wird der Parameter "actionData" mit dem actionData-Wert des Buttons belegt.
- Der zwei- oder mehrmalige Aufruf von TimerStop mit dem gleichen TimerHandle oder mit einem Null-Handle (leeres Handle) ist erlaubt. Sie können deshalb in Ihrem OnExit-Handler einfach sämtliche Timer stoppen (Aufruf von TimerStop), egal ob die Timer noch laufen oder ob sie je benutzt wurden (unbenutzte Handles sind leere Handles).

17 Arbeit mit der Maus

17.1 Überblick

Die meisten GEOS-Objekte können automatisch auf Mausereignisse reagieren. So weiß ein Button was zu tun ist, wenn er mit der Maus angeklickt wird. Hier braucht und kann der R-BASIC Programmierer nicht eingreifen. In vielen Fällen muss der R-BASIC Programmierer jedoch die Reaktion auf ein Mausereignis selbst behandeln. Deshalb gibt es eine Reihe von Objekten, die eine explizite Mausunterstützung anbieten. Das sind die Objekte:

VisContent, BitmapContent, VisObj, Canvas und Image

Dazu sind die folgenden Actionhandler, Instancevariablen und Methoden definiert. Details dazu, insbesondere unter welchen Bedingungen die entsprechenden Handler gerufen werden, finden Sie in den nächsten Abschnitten.

Action-Handler-Typen:

Handler-Typ	Parameter
MouseAction	(sender as object, xPos, yPos, event as Integer)

Spezielle Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
OnMouseButton	OnMouseButton = **<Handler>**	nur schreiben
OnMouseMove	OnMouseMove = **<Handler>**	nur schreiben
OnMouseOver	OnMouseOver = **<Handler>**	nur schreiben
sendMouseEvents	sendMouseEvents = **bits** [, **mode**]	lesen, schreiben

Methoden:

Methode	Aufgabe
GrabMouse	Objekt "greift sich" die Maus
ReleaseMouse	Objekt lässt die Maus wieder los
TestInside	Prüft ob ein Koordinatenpaar im Objektbereich liegt
TestInsideAC	Prüft ob ein Koordinatenpaar im Objektbereich liegt

Ein einfaches Beispiel

```
Canvas MyMouseObj
  OnMouseButton = ButtonPressed
  ...
End Object
```

```
MOUSEACTION ButtonPressed
  IF event = ME_LEFT_DOWN THEN MsgBox("Linke Maustaste gedrückt")
End Action
```

17.2 Maus Grabbing

Die folgenden Aussagen gelten nicht für VisContent und BitmapContent Objekte.

Im Normalfall wird ein Mausereignis von einem zum nächsten Objekt weitergereicht, bis es das Objekt erreicht hat, das sich direkt unter dem Mauszeiger befindet. Einige Ereignisse, insbesondere das Loslassen der Maustasten und das Verlassen des Objektbereichs, werden im Normalfall gar nicht durchgestellt.
Es ist deshalb oft erforderlich, dass ein Objekt die Mausereignisse exklusiv und komplett zu sehen bekommt. Dieser Prozess heißt "Grabbing". Das Objekt "greift" sich die Mausereignisse direkt und ohne Umweg. Dafür stehen die folgenden Methoden zur Verfügung:

Methode	Aufgabe
GrabMouse	Objekt "greift sich" die Maus
ReleaseMouse	Objekt lässt die Maus wieder los

Syntax im BASIC Code: **<obj>.GrabMouse**
<obj>.ReleaseMouse

Das Objekt zählt nicht mit, wie oft es die Maus gegrabbt hat. Grabbt ein Objekt die Maus mehrfach so wird trotzdem bei ersten ReleaseMouse die Maus wieder freigegeben.
Durch das Grabben der Maus wird erreicht, dass das System alle Mausereignisse direkt an das Objekt sendet, unabhängig davon, wo sich der Mauspointer gerade befindet. Das bedeutet insbesondere, dass das Objekt auch dann die Mausereignisse erhält, wenn sich der Mauszeiger nicht mehr über dem Objekt befindet.
Das Objekt bekommt solange alle Mausereignisse, bis es die Maus wieder freigibt. In der Zwischenzeit ist weder die Bedienung von Menüs möglich noch kann sich ein anderes Objekt die Maus grabben. Häufig ist es so, dass sich ein Objekt beim Drücken einer Maustaste die Maus grabbt und sie beim Loslassen der Taste wieder freigibt. Entsprechende Beispiele finden Sie in den nächsten Kapiteln.

VisContent und BitmapContent

VisContent und BitmapContent Objekte können die Maus nicht grabben. Die Methoden GrabMouse und ReleaseMouse sind wirkungslos. Im Gegenzug erhalten diese Objekte stets alle Mausevents, so dass ein Grabbing der Maus gar nicht nötig ist. Sobald eine Maustaste über einem VisContent- oder BitmapContent gerückt wird grabbt das Objekt die Maus implizit, d.h. alle Mausereignisse gehen an dieses Objekt, auch wenn der Mauszeiger das zugehörige View zwischenzeitlich verlässt. Im Unterschied zum expliziten Grabbing gehen die Mausereignisse zusätzlich an eventuelle Children des VisContent-Objekts. Diese Children (vom Typ VisObj) können dann die Maus explizit grabben.

17.3 Aufruf der Actionhandler

Die Mausereignisse sind in drei Gruppen eingeteilt, für die es jeweils einen eigenen Actionhandler gibt.

Handler	Zugeordnete Ereignisse
OnMouseButton	Linke Maustaste wird gedrückt Doppelklick mit der linken Maustaste Linke Maustaste wird gehalten (G) Linke Maustaste wird losgelassen (G) Rechte Maustaste wird gedrückt Doppelklick mit der rechten Maustaste Rechte Maustaste wird gehalten (G) Rechte Maustaste wird losgelassen (G)
OnMouseMove	Maus bewegt sich im Objektbereich
OnMouseOver	Maus "betritt" den Objektbereich Maus "verlässt" den Objektbereich (G)

Die genannten Ereignisse führen nur dann zum Aufruf des Actionhandlers, wenn die folgenden Bedingungenerfüllt sind

- Es ist ein Handler zugewiesen.
- Das entsprechende Bit ist in der Instancevariablen **sendMouseEvents** gesetzt.
- Bei den mit (G) gekennzeichneten Ereignissen: Das Objekt hat die Maus "gegrabbt".
  Diese Beschränkung gilt nicht für VisContent und BitmapContent. Für sie gelten nur die ersten beiden Bedingungen.

Syntax UI- Code:	**OnMouseButton** = **<Handler>**
	OnMouseMove = **<Handler>**
	OnMouseOver = **<Handler>**
Schreiben:	**<obj>**.**OnMouseButton** = **<Handler>**
	<obj>.**OnMouseMove** = **<Handler>**
	<obj>.**OnMouseOver** = **<Handler>**

Durch Zuweisen des speziellen "Handlers" **NoAction** kann man die Zuweisung eines Handlers aufheben, z.B.

```
sender.OnMouseMove = NoAction
```

NoAction kann mit allen Handlern, nicht nur mit Maushandlern, benutzt werden.

Alle Mausaction Handler haben die folgenden Parameter:

sender:	Das Objekt, welches das Mausereignis ausgelöst hat
xPos, yPos:	Die Koordinaten des Mauszeigers, relativ zur linken oberen Ecke des Objekts.
event:	Information welches Ereignis zum Aufruf geführt hat.

Für event sind folgenden Konstanten definiert.

Konstante	Wert	Bedeutung
ME_LEFT_DOWN	1	Linke Maustaste wird gedrückt
ME_LEFT_UP	2	Linke Maustaste wird losgelassen
ME_LEFT_DOUBLE	4	Doppelklick links
ME_LEFT_HOLD	8	Linke Maustaste wird gedrückt gehalten
ME_RIGHT_DOWN	16	Rechte Maustaste wird gedrückt
ME_RIGHT_UP	32	Rechte Maustaste wird losgelassen
ME_RIGHT_DOUBLE	64	Doppelklick rechts
ME_RIGHT_HOLD	128	Rechte Maustaste wird gedrückt gehalten
ME_MOVE	256	Maus bewegt sich im Objektbereich
ME_ENTER	512	Maus "betritt" den Objektbereich
ME_LEAVE	1024	Maus "verlässt" den Objektbereich

Anmerkungen

Doppelklicks:

Drückt der Nutzer z.B. die linke Maustaste erstmalig so wird zunächst ein LEFT_DOWN-Ereignis erzeugt. Lässt der Nutzer die Taste kurz darauf los und drückt sie ein zweites Mal (dh. er führt einen Doppelklick aus) so wird zusätzlich ein LEFT_DOUBLE-Ereignis erzeugt.

Hold-Ereignisse

Drückt der Nutzer z.B. die linke Maustaste so wird zunächst wieder ein LEFT_DOWN-Ereignis erzeugt. Hält er jetzt die Maustaste für eine bestimmte Zeit gedrückt (ca. 0,5 Sekunden) so wird zusätzlich ein LEFT_HOLD-Ereignis erzeugt. Sie können damit z.B. unterscheiden ob der Nutzer etwas nur anklicken oder es festhalten und bewegen will.

sendMouseEvents

Die Instancevariable sendMouseEvents bietet die Möglichkeit die Behandlung von Mausereignissen schnell ein- und auszuschalten. Das ist wesentlich effektiver als jedes Mal den entsprechenden Handler aufzurufen und dort abzufragen, ob er aktuell auch erwünscht ist. Insbesondere der OnMouseMove Handler wird, wenn sich die Maus über dem Objekt befindet, extrem häufig gerufen. Das kann zu einer unerwünschten Belastung und zur Verlangsamung des Systems führen.

SendMouseEvents ist ein Bitfeld, d.h. jedes Bit hat eine bestimmte Bedeutung. Die Bits sind identisch mit den ME_-Werten aus der Tabelle oben.

Hinweis: Per Default sind alle Bits gesetzt, die sollten daher im UI-Code immer genau die Bits (ME_-Werte) setzen, die Sie benötigen. Nicht in der Tabelle aufgeführte Bits sind reserviert und sollten Null bleiben.

Syntax UI-Code: **sendMouseEvents** = **bits** [, **mode**]
bits: Kombination der ME_-Werte (siehe oben)
mode: bestimmt, wie der übergebene Bit-Wert zu behandeln ist (siehe Tabelle unten)
Defaultwert: 0 (REPALACE_BITS)

Lesen: **<numVar>** = **<obj>.sendMouseEvents (0)**
Die BASIC Syntax verlangt beim Lesen von sendMouseEvents einen Parameter, weil sendMouseEvents beim Schreiben zwei Parameter hat. Der in der Klammer stehende Wert wird jedoch ignoriert.

Schreiben: **<obj>.sendMouseEvents** = **bits [, mode]**

Für Mode sind die folgenden Konstanten definiert:

Konstante	Wert	Bedeutung
REPLACE_BITS	0	Der Wert wird 1:1 zugewiesen, d.h. der Instancewert wird, wie bei einer normalen Zuweisung, mit dem neuen Wert überschrieben. Das ist der Defaultwert und wird meist im UI-Code benutzt.
SET_BITS	1	Angegebene Bits auf den Wert 1 setzen. Die anderen Bits werden nicht beeinflusst. Verwenden Sie diesen Mode-Wert wenn Sie die Behandlung einen Ereignisses aktivieren wollen.
CLEAR_BITS	2	Angegebene Bits auf den Wert 0 setzen. Die anderen Bits werden nicht beeinflusst. Verwenden Sie diesen Mode-Wert wenn Sie die Behandlung einen Ereignisses ausschalten wollen.

Beispiele

```
' Ein und Ausschalten der Mauszeiger-Ereignisse
  MyMouseObj.sendMouseEvents = ME_MOVE, SET_BITS
  MyMouseObj.sendMouseEvents = ME_MOVE, CLEAR_BITS
```

```
' Lesen des sendMouseEvents-Werts und Prüfen, ob ME_MOVE
' gesetzt ist
DIM bits as WORD
  bits = MyMouseObj.sendMouseEvents ( 0 )
  IF bits AND ME_MOVE  THEN  MsgBox "Ja"
```

```
' Prüfen ob ME_MOVE ODER ME_LEFT_UP gesetzt ist
  IF bits AND ( ME_MOVE OR ME_LEFT_UP) THEN  MsgBox "Ja"
```

17.4 Typische Situationen

Dieser Abschnitt beschreibt die Behandlung typischer Fälle, die beim Arbeiten mit der Maus auftreten können.

17.4.1 Behandlung der Mousebuttons

In diesem Abschnitt wird am Beispiel der linken Maustaste das typische Vorgehen für diesen Fall erklärt.

Im UI-Code sollten wir nur die Bits im Feld sendMouseEvents setzen, die wir auch wirklich benötigen. Das sind ME_LEFT_DOWN und ME_LEFT_UP, alle anderen Mausereignisse werden dann vom Objekt ignoriert.

```
Canvas MyOutputObj
  ....
  sendMouseEvents = ME_LEFT_DOWN + ME_LEFT_UP
  OnMouseButton = ButtonPressed
End Object
```

Im Actionhandler unterscheiden wir mit einer On - SWITCH Anweisung zwischen den beiden Ereignissen. Außerdem müssen wir beim Drücken der linken Maustaste die Maus "grabben" (GrabMouse), sonst wird das Ereignisse "Loslassen" nicht an den Handler weitergeleitet. Das ist eine GEOS-intere Optimierungsfunktion. Entsprechend müssen wir beim Loslassen die Maus wieder freigeben (ReleaseMouse)

```
MOUSEACTION ButtonPressed
   ON event SWITCH

  CASE ME_LEFT_DOWN:
    sender.GrabMouse
    '<... hier Aktionen ausführen ...>
    End CASE

  CASE ME_LEFT_UP:
    sender.ReleaseMouse
    '<.. hier Aktionen ausführen ...>
    End CASE

  End SWITCH

End Action
```

Statt des LEFT_DOWN-Ereignisses könnte man auch das LEFT_HOLD-Ereignis abfragen. Das hätte den Vorteil, dass ein kurzer - eventuell versehentlicher - Klick wirkungslos bleibt. Entsprechende Beispiele finden Sie im nächsten Kapitel.
Bei entsprechender Programmierung kann man auch zwischen einem Einfachklick, einem Doppelklick und einem längeren Festhalten (HOLD) des Mausbuttons unterscheiden.

17.4.2 Arbeit mit dem OnMouseMove Handler

Wenn Sie während der Bewegung des Mauszeigers bestimmte Aktionen auslösen wollen (z.B. etwas zeichnen, siehe nächster Abschnitt) müssen Sie einen OnMouseMove Handler benutzen. Der Handler wird gerufen, sobald sich die Maus über dem Objekt bewegt. Da das in schneller Folge passiert ist es sinnvoll, diesen Handler nur freizuschalten, wenn er gebraucht wird. Ein sehr häufiger Fall ist, dass er nur benötigt wird, während die linke Maustaste gedrückt ist. Diese Situation wird im Folgenden beschrieben.

Im UI-Code setzen wir in sendMouseEvents nur die Bits ME_LEFT_HOLD und ME_LEFT_UP, d.h. der OnMouseMove Handler ist zunächst inaktiv, weil wir das Bit ME_MOVE **nicht** setzen.
Ob wir zum Starten der Aktion das Ereignis ME_LEFT_DOWN oder ME_LEFT_HOLD verwenden hängt von der konkreten Situation und von den Intentionen des Programmierers ab. In einem Zeichenprogramm wird häufig ME_LEFT_DOWN bevorzugt während zum Bewegen von Objekten über den Schirm ME_LEFT_HOLD der Vorzug gegeben wird.
Der Code verwendet ein VisObj, er ist aber genauso auf jedes andere Objekt, dass die Maus unterstützt, anwendbar.

```
VisObj MyObj
  ....
  sendMouseEvents = ME_LEFT_HOLD + ME_LEFT_UP
  OnMouseButton = ButtonPressed
  OnMouseMove = MoveIt
End Object
```

Im Actionhandler "ButtonPressed" müssen wir zusätzlich zum Beispiel aus dem vorherigen Kapitel noch den OnMouseMove Handler freischalten bzw. deaktivieren. Der Parameter SET_BITS sorgt dafür, dass das Bit ME_MOVE gesetzt wird, alle anderen Bits aber nicht geändert werden. Analog sorgt CLEAR_BITS dafür, dass das Bit ME_MOVE zurückgesetzt (auf Null gesetzt) wird, ohne dass die anderen Bits beeinflusst werden.

```
MOUSEACTION ButtonPressed
  ON event SWITCH

  CASE ME_LEFT_HOLD:
    sender.GrabMouse
    sender.sendMouseEvents = ME_MOVE, SET_BITS
    '<... hier weitere Aktionen ausführen ...>
    End CASE

  CASE ME_LEFT_UP:
    sender.ReleaseMouse
    sender.sendMouseEvents = ME_MOVE, CLEAR_BITS
    '<..  hier weitere Aktionen ausführen ...>
    End CASE
  End SWITCH
End Action
```

Der OnMouseMove Handler "MoveIt" wird nun nur gerufen, während die linke Maustaste gedrückt ist. Die Anweisung "sender.GrabMouse" sorgt im Übrigen auch dafür, dass dieser Handler auch dann gerufen wird, wenn der Nutzer die Maus aus dem Objekt herausbewegt. Das ist sehr praktisch, da wir diesen Fall dann nicht extra behandeln müssen. Das Grafiksystem von GEOS sorgt dabei dafür, dass wir nicht über den Rand des zugehörigen Views (bzw. bei Canvas und Image Objekten nicht über den Rand des Objekts) malen können.

```
MOUSEACTION MoveIt
  '<... hier Aktionen ausführen ...>
End Action
```

17.4.3 Zeichnen auf den Bildschirm

Sehr häufig wollen wir mit der Maus etwas auf den Bildschirm zeichnen. Dieser Abschnitt beschreibt, wie man dazu vorgehen muss. Wir setzen den folgenden UI-Code voraus:

```
BitmapContent MyBitmap
  ....
  sendMouseEvents = ME_LEFT_HOLD + ME_LEFT_UP
  OnMouseButton = ButtonPressed
  OnMouseMove = MoveIt
End Object
```

Beim Drücken der Linken Maustaste müssen wir jetzt unser Objekt zum "Screen" machen. Damit gehen alle Grafik- und Textausgaben an dieses Objekt und erscheinen somit auf dem Bildschirm. Analog müssen wir den Screen wieder zurücksetzen, wenn die Maustaste losgelassen wird. Außerdem müssen wir unbedingt den vorher aktiven Screen in einer (globalen) Variablen speichern.
Beim BitmapContent (wie im Beispiel) gehen die Grafik- und Textausgaben parallel dazu in die Bitmap, so dass sie automatisch "gespeichert" werden. Bei anderen Objekten, wie z.B. einem Canvas oder einem VisObj gehen sie nur auf den Bildschirm und wir müssen selbst dafür sorgen, dass unser "Ergebnis" auf geeignete Weise gespeichert wird.
Weil wir ein BitmapContent verwenden brauchen wir die Maus nicht zu Grabben und zu Releasen. Sobald eine Maustaste über einem VisContent- oder BitmapContent gerückt und festgehalten wird gehen alle Mausereignisse an dieses Objekt, auch wenn der Mauszeiger das zugehörige View zwischenzeitlich verlässt.

```
DIM oldScreen as Object       ' Zur Veranschaulichung rot

MOUSEACTION ButtonPressed
   ON event SWITCH

  CASE ME_LEFT_HOLD:
    sender.sendMouseEvents = ME_MOVE, SET_BITS
    oldScreen = Screen
    Screen = sender
    '< ... mehr ..>
    End CASE

  CASE ME_LEFT_UP:
    sender.sendMouseEvents = ME_MOVE, CLEAR_BITS
    Screen = oldScreen
    '< ... mehr ..>
    End CASE

  End SWITCH

End Action
```

Zur Demonstration wollen wir eine "Gummilinie" implementieren. Dazu benötigen wir globale Variablen. X0 und y0 sind der Startpunkt, x1, und y1 der sich mit der Mausbewegung ändernde Endpunkt der Linie. OldScreen soll den beim Aufruf des Maushandlers aktiven Screen zwischenspeichern.

```
DIM x0, y0, x1, y1
DIM oldScreen as Object        ' Zur Veranschaulichung rot
```

Beim Drücken der linken Maustaste müssen wir die Linienkoordinaten auf die aktuelle Mausposition setzen. Wir stellen den MixMode MM_INVERT ein, der wie für eine Gummilinie gemacht ist. In diesem Modus werden die Pixels, auf die Linien und andere Grafikausgaben wirken, nicht mit der Vordergrundfarbe überschrieben, sondern die Farbcodes werden invertiert. Das hat zwei Konsequenzen. Erstens ist die Linie immer zusehen, egal welche Farbe der Hintergrund hat und zweitens bewirkt ein zweimaliges Zeichnen der gleichen Linie, dass sie wieder verschwindet. Das ist genau das, was wir brauchen.
Schließlich stellen wir noch eine Linienbreite von 8 Pixeln ein und zeichnen sie erstmalig. In unserem Fall ergibt das nur einen Punkt auf dem Schirm, der aber nötig ist, weil der OnMove-Handler ihn wieder löscht.
Der fertige Codeabschnitt sieht also so aus:

```
  CASE ME_LEFT_HOLD:
    sender.GrabMouse
    sender.sendMouseEvents = ME_MOVE, SET_BITS
    oldScreen = Screen
    Screen = sender
    x0 = xPos : y0 = yPos
    x1 = xPos : y1 = yPos
    graphic.MixMode = MM_INVERT
    graphic.lineWidth = 8
    Line x0, y0, x1, y1
    End CASE
```

Der OnMove Handler hat nun nur wenig zu tun. Unser Objekt ist noch der Screen. Die erste Line Anweisung löscht die aktuelle Line vom Bildschirm. Das geht, weil wir den MixMode MM_INVERT eingestellt haben. Dann speichern wir die neuen Endkoordinaten und zeichnen die Line erneut.

```
MOUSEACTION MoveIt
  Line x0, y0, x1, y1
  x1 = xPos
  y1 = yPos
  Line x0, y0, x1, y1
End Action
```

Etwas schneller - und damit weniger anfällig gegen Flackern - wäre folgende Sequenz:

```
  Line x0, y0, x1, y1
  Line x0, y0, xPos, yPos      ' Koordinaten beachten!
  x1 = xPos
  y1 = yPos
```

Beim Loslassen der Maustaste löschen wir zuerst die vorhandene Linie, stellen den "normalen" MixMode MM_SET ein und zeichnen die Line dann permanent in weißer Farbe. Schließlich setzen wir den Screen zurück.

```
  CASE ME_LEFT_UP:
    sender.ReleaseMouse
    sender.sendMouseEvents = ME_MOVE, CLEAR_BITS
    Line x0, y0, x1, y1
    graphic.MixMode = MM_SET
    INK WHITE
    Line x0, y0, x1, y1
    Screen = oldScreen                    ' nicht vergessen!
    End CASE
```

Wichtige Hinweise:

Wir müssen der Zuweisung der globalen Screen-Variablen große Aufmerksamkeit widmen, insbesondere wenn Sie mehrere Objekte haben, die Grafik ausgeben können (z.B. ein VisContent und/oder mehrere VisObj-Objekte). Deswegen der scheinbar umständliche Weg mit der globalen Variablen oldScreen. Vergessen wir das "Zurücksetzen" des Screens kann GEOS crashen - entweder gleich oder beim Beenden des Programms.

Außerdem sollten wir uns in einer globalen Variablen merken, dass die Maustaste gedrückt ist und dies im OnMove Hander abfragen. Der Grund ist, dass nach dem Loslassen der Maustaste noch ein OnMove Event in der Wartschlange sein **könnte**, dass dann auf den falschen Screen zeichnet. Die folgenden Codezeilen sind an den entsprechenden Stellen einzufügen.

```
DIM mouseDown             ' globale Variable

' Im ME_LEFT_HOLD-Zweig:
  mouseDown = TRUE

' Im ME_LEFT_UP-Zweig:
  mouseDown = FALSE

' Im MoveIt Handler:
  IF mouseDown = FALSE THEN RETURN
```

Weitere Beispiele zum Thema Maushandling finden Sie bei der Beschreibung der Objekte VisContent und VisObj.

17.4.4 Behandeln von MouseOver Ereignissen

Gelegentlich ist es sinnvoll einfach nur zu wissen, ob sich die Maus über dem vom Objekt eingenommen Bildschirmbereich befindet oder nicht. Diesem Zweck dient der OnMouseOver Handler. Er wird gerufen, wenn die Maus den Bereich des Objekts betritt (event = ME_ENTER) oder ihn verlässt (event = ME_LEAVE).

Beachten Sie, dass das Ereignis "Verlassen des Objektsbereichs" nur gesendet wird, wenn das Objekt die Maus gegrabbt hat. Ausnahmen sind die Objekte VisContent und BitmapContent. Sie senden dieses ME_LEAVE-Ereignis in jedem Fall.

Eine typische Implementation könnte also wie folgt aussehen. Das Objekt stellt eine gelbe Ellipse dar, die rot wird, wenn sich der Mauszeiger über dem Objekt befindet.

UI-Code:

```
Canvas Area
  fixedSize = 200, 100
  sendMouseEvents = ME_ENTER + ME_LEAVE
  OnMouseOver = OverHandler
  OnDraw = DrawHandler
End Object
```

BASIC-Code:

```
DrawAction DrawHandler
  Rectangle 0, 0, MaxX, MaxY, Black
  Fillellipse 2, 2, MaxX-2, MaxY-2, Yellow
End Action
```

Im Handler für den Mauszeiger machen wir zunächst das das Canvas-Objekt zum Screen, grabben uns die Maus (damit das ME_LEAVE-Ereignis gesendet wird) und zeichnen dann eine rote Ellipse. Beim Verlassen des Objektbereichs machen wir die Ellipse wieder gelb, setzen den Screen zurück und geben die Maus wieder frei.

```
MouseAction OverHandler
  ON event Switch
  case ME_ENTER
    Screen = sender
    Sender.GrabMouse
    FillEllipse 2, 2, MaxX-2, MaxY-2, RED
    End CASE
  case ME_LEAVE
    FillEllipse 2, 2, MaxX-2, MaxY-2, YELLOW
    Screen = NullObj()
    Sender.ReleaseMouse
    End CASE
  End SWITCH
End Action
```

Das nächste Beispiel für einen OnMouseOver-Handler gibt eine Information an ein Textobjekt aus, je nachdem, ob sich die Maus über dem Objekt befindet oder nicht.

```
MouseAction OverHandler
  ON event Switch
  case ME_ENTER
    Sender.GrabMouse
    InfoText.text$ = "Maus ist über dem Objekt"
    End CASE
  case ME_LEAVE
    InfoText.text$ = ""
    Sender.ReleaseMouse
    End CASE
  End SWITCH
End Action
```

Hinweise / Technische Details

1. Um zu erkennen, ob der Mauszeiger gerade den Objektbereich betritt oder verlässt muss sich das Objekt merken, ob der Mauszeiger vorher innerhalb oder außerhalb der Grenzen des Objekts war. Die Information, dass sich der Mauszeiger außerhalb der Grenzen des Objekts befindet bekommt es aber nur, wenn es die Maus gegrabbt hat. Unterlassen Sie das Grabben der Maus beim Betreten des Objekts, so erkennt das Objekt nicht mehr, wenn der Mauszeiger seine Grenzen verlässt. Ein erneutes Betreten des Objektbereichs wird daher auch nicht erkannt und die entsprechende Message wird nicht noch einmal gesendet.
   Ausnahmen sind hier wieder das VisContent und das BitmapContent Objekt. Sie erhalten die Information "Objektbereich verlassen" bzw. ".. betreten" in jedem Fall.
2. Bewegt der Nutzer die Maus so schnell, dass sie aus dem Objektbereich direkt in das Fenster eines anderen Programms springt (ohne dass noch ein Mausereignis innerhalb des eigenen Programms erzeugt wird), so wir das Ereignis "Objektbereich verlassen" zunächst nicht gesendet. Es wird stattdessen gesendet, wenn der Mauszeiger das Hauptfenster unseres Programms wieder betritt. Das ist kein Fehler von R-BASIC, sondern eine Eigenschaft des GEOS-Systems.
3. Die Parameter xPos und yPos des OnMouseOver-Handlers enthalten für die Objekte VisContent und BitmapContent stets den Wert Null. Für alle anderen Objekte enthalten Sie die Koordinaten, bei denen das Objekt betreten bzw. verlassen wurde. Das ist im Allgemeinen dicht am Rand des Objekts.

17.4.5 Abfrage der Tastatur

Gelegentlich muss man in einem Maushandler unterschiedliche Operationen auslösen, je nachdem, ob gleichzeitig eine bestimmte Taste auf der Tastatur gedrückt ist oder nicht. Insbesondere die Steuertasten wie Shift, Ctrl (Strg) usw., die mit **GetKeyState** abgefragt werden können, sind hier interessant.

GetKeyState liefert einen Word-Wert, dessen einzelne Bits die folgende Bedeutung haben:

Konstante (Shift-State)	Wert (hex.)	Bedeutung
–	1 &h01	Feuertaste 1 am Joystick
–	2 &h02	Feuertaste 2 am Joystick
KS_RSHIFT	4 &h04	Rechte Shift-Taste
KS_LSHIFT	8 &h08	Linke Shift-Taste
KS_RCTRL	16 &h10	Rechte Strg-Taste
KS_LCTRL	32 &h20	Linke Strg-Taste
KS_RALT	64 &h40	Rechte Alt-Taste
KS_LALT	128 &h80	Linke Alt-Taste

Konstante (Toggle-State)	Wert	Bedeutung
KS_SCROLL_LOCK	256 (&h100)	Scroll-Lock-Taste (Rollen) eingerastet
KS_NUM_LOCK	512 (&h200)	Num-Lock-Taste eingerastet
KS_CAPS_LOCK	1024 (&h400)	Shift-Lock Taste eingerastet

Zur Abfrage der Bits muss man die logische AND Funktion verwenden:

```
'Abfrage ob eine Shift-Taste gedrückt ist
IF GetKeyState AND ( KS_LSHIFT OR KS_RSHIFT ) THEN ....

' Abfrage ob die NUM-Lock Taste gedrückt ist
IF GetKeyState AND  KS_NUM_LOCK  THEN ....
```

Weitere Informationen zu GetKeyState finden Sie im Programmier-Handbuch.

Zur Tastaturabfrage innerhalb von Maushandlern eignen sich außerdem die folgenden Anweisungen bzw. globale Variablen:

GetKey

GetKeyLP — Globale Variablen, die die aktuell bzw. zuletzt gedrückte Taste enthalten. Das kann ein ASCII-Code oder bei Steuertasten wie F1 ein erweiterter Code (> 256) sein.

Inkey$ — Liest ein einzelnes Zeichen von der Tastatur.

Details dazu finden Sie an den entsprechenden Stellen im Programmier-Handbuch. Alternativ können Sie auch den Tastaturhandler des Objekts benutzen, falls es einen besitzt, um über den Zustand der Tastatur auf dem Laufenden zu sein. Bitte benutzen Sie nicht die Anweisungen Input bzw. InputBox. Das kann zu Konflikten oder zu unerwartetem Verhalten führen.

17.5 Utility Methoden

Methode	Aufgabe
TestInside	Prüft ob ein Koordinatenpaar im Objektbereich liegt
TestInsideAC	Prüft ob ein Koordinatenpaar im Objektbereich liegt

Syntax im BASIC Code: **<z>** = **<obj>**.**TestInside** (x, y)
<z> = **<obj>**.**TestInsideAC** (x, y)
<z>: numerische Variable
Return: z ist Null, wenn das Koordinatenpaar innerhalb des Objekts liegt
z ist größer als Null, wenn nicht

Die beiden Methoden prüfen, ob ein Koordinatenpaar im vom Objekt überdeckten Bildschirmbereich liegt oder nicht. **TestInside** setzt voraus, dass die linke obere Ecke des Objekts die Koordinaten (0; 0) hat. Das ist das gleiche Koordinatensystem, dass verwendet wird, wenn das Objekt der Screen ist (Vergleiche 17.4.3 Zeichnen auf den Bildschirm). Das heißt, die x-Koordinate liegt außerhalb des Objekts wenn gilt: x < 0 oder x > object.xSize. Analoges gilt für die y-Koordinate.
TestInsideAC (AC = absolute coordiates, absolute Koordinaten) berücksichtigt die Position des Objekts innerhalb des übergeordneten Fensters. Das heißt, die x-Koordinate liegt außerhalb des Objekts wenn gilt: x < object.xPosition oder x > (object.xPosition + object.xSize). Analoges gilt für die y-Koordinate.
TestInside bzw. TestInsideAC sind dabei sehr viel schneller als die manuelle Abfrage der Positionen entsprechend den obigen Beziehungen.
Außerdem enthält der zurückgelieferte Wert Informationen darüber, wo genau sich das Koordinatenpaar relativ zum Objekt befindet. Das Bild rechts veranschaulicht das. Der Wert ist die Summe aus folgenden Informationen:

3	1	5
2	0	4
10	8	12

1: Die y-Koordinate liegt oberhalb des Objekts
2: Die x-Koordinate liegt links vom Objekt
4: Die x-Koordinate liegt rechts vom Objekt
8: Die y-Koordinate liegt unterhalb des Objekts

Liegt das Koordinatenpaar links oberhalb des Objekts beträgt der zurückgelieferte Wert 1 + 2 = 3.
Diese Informationen können mit der logischen Operation AND abgefragt werden. Der folgende Code fragt ob, ob die x-Koordinate rechts vom Objekt liegt. Die Print-Anweisung wird also für die Fälle 5, 4 und 12 ausgeführt:

```
z = MyObj.TestInside(x, y)
IF z AND 4 THEN Print "Rechts vom Objekt"
```

Wenn Sie nur die Information benötigen, ob sich der Mauszeiger innerhalb oder außerhalb des Objekts befindet können Sie auch einen OnMouseOver-Handler verwenden und die Information in einer globalen Variablen speichern. Beachten Sie, dass Sie dazu gegebenenfalls die Maus grabben müssen.

(Leerseite)

18 Abwärtskompatibilität

18.1 Der klassische BASIC Modus

Ältere, nicht objekt-orientierte BASIC Programme besitzen ein "Hauptprogramm", das beim Start des Programms automatisch ausgeführt wird. Außerdem gibt es einen vordefinierten Bildschirm auf die alle Text und Grafikausgaben erfolgen. R-BASIC kann dieses Verhalten simulieren, so dass das Übertragen älterer Programme nach R-BASIC vereinfacht wird. Dazu gibt es die Anweisung ClassicCode.

Syntax: **ClassicCode**

ClassicCode muss vor der ersten ausführbaren Anweisung (z.B. Print) im Quelltext stehen. Fall Sie Include-Anweisungen verwenden muss ClassicCode nach den Include-Anweisungen stehen.

Die Anweisung ClassicCode bewirkt folgendes:
- Das Scheiben von Code außerhalb von Routinen (das klassische Hauptprogramm) wird zugelassen.
- Dieser "klassische" Code wird beim Start des Programms automatisch ausgeführt.
- R-BASIC legt beim ersten Start des klassischen Programms ein paar Objekte an, damit Text und Grafik-Anweisungen arbeiten können. Konkret sind das ein Primary mit einem View und einem BitmapContent Objekt der Größe 640x400 Pixel mit 256 Farben. Einmal angelegt können Sie diese UI-Objekte ändern oder ergänzen, wenn Sie wollen.

Hinweis: Auch unter objekt-orientierten Programmen gibt es das Problem, dass beim Start des Programms automatisch Code ausgeführt werden muss. Die Lösung für dieses Problem unter R-BASIC ist, einen OnStartup Handler für das Application-Objekt zu programmieren. Details dazu finden Sie im Objekthandbuch, bei der Beschreibung des Application Objekts (Kapitel 4.1).

18.2 Zeilennummern

Viele ältere BASIC-Programme lesen sich etwa so:

```
10  Print "Bitte geben Sie eine Zahl ein"
20  INPUT  Z
30  IF Z < 5 THEN PRINT "Unter Fünf"
40  GOTO  10
```

Die Zahlen am Beginn jeder Zeile werden als Zeilennummern bezeichnet und sind bei älteren BASIC-Interpretern lebensnotwendig. R-BASIC kommt gänzlich ohne Zeilennummern aus, zur Definition von Sprungzielen gibt es den LABEL Befehl.

Achtung! Verwechseln Sie diese im Code vereinbarten Zeilennummern nicht mit den im Editorfenster angezeigten Zeilennummern!

R-BASIC kann jedoch mit Zeilennummern umgehen (Bei GOTO, GOSUB und RESTORE), so dass ältere Programme einfacher nach R-BASIC zu übertragen sind. Dazu müssen Sie die Zeilennummern aber explizit im Programm vereinbaren. Beispielsweise könnten Sie obiges Programm direkt so eingeben, wie es oben steht. Es geht aber auch folgende Variante:

```
10 Print "Bitte geben Sie eine Zahl ein"
  INPUT  Z
  IF Z < 5 THEN PRINT "Unter Fünf"
  GOTO  10
```

Wesentlich besser ist aber diese Form:

```
Label restart
  Print "Bitte geben Sie eine Zahl ein"
  INPUT  Z
  IF Z < 5 THEN PRINT "Unter Fünf"
  GOTO  restart
```

Hinweis: R-BASIC prüft die Gültigkeit von Zeilennummern nicht! Mehrfach definierte Zeilennummern werden nicht erkannt! Die Verwendung von Zeilennummern, die nicht existieren (also nicht explizit angegeben wurden) führt aber zu einem Compilerfehler.

Die Verwendung von Zeilennummern ist mit den Befehlen GOTO, GOSUB und RESTORE möglich.

Syntaxvariante	Beispiel
GOTO Zeilennummer	**GOTO** 1000
ON **<Ausdruck> GOTO <Zeilennummern>**	ON x+5 **GOTO** 10, 20, 30
GOSUB Zeilennummer	**GOSUB** 1000
ON **<Ausdruck> GOSUB <Zeilennummern>**	ON x-1 **GOSUB** 790, 20, 50
RESTORE Zeilennummer	**RESTORE** 240

18.3 Kompatibilität mit dem KC-85-BASIC

Das folgende Kapitel ist für Nutzer interessant, die einen KC-85 Homecomputer besitzen oder besessen haben oder aus einem anderen Grund ein KC-85 Programm nach R-BASIC portieren wollen. R-BASIC ist weitgehend abwärtskompatibel zum BASIC des KC-85/3 und KC-85/4, so dass reine KC-BASIC-Programme ohne größere Probleme laufen sollten.

Hinweise zur Portierung von KC-85 BASIC Programmen

Erforderliche Einstellungen

R-BASIC ist nicht als KC-Emulator konzipiert. Insbesondere steht die Hardware des KC-85 nicht zur Verfügung und wird auch nicht emuliert. Mit der Systemvariablen **kc85Features** können Sie R-BASIC trotzdem dazu bringen, sich in bestimmten Situationen wie der KC-85 zu verhalten. In diesem Modus können Sie weiterhin alle R-BASIC Befehle verwenden. Ausnahme sind einige Farbbefehle, da auf dem KC-85 nur maximal 16 Farben möglich sind.
Zusätzlich müssen Sie bestimmte Bedingungen einhalten. Dazu gehört, dass Sie als Screen eine 8Bit Bitmap mit aktivierter Palette (idealer Weise 640x512 Pixel) verwenden sowie das Koordinatensystem drehen. Wie man das macht und weitere notwendige Einstellungen finden Sie in der Beispieldatei "KC 85 Demo" im Ordner "R-BASIC\Beispiel\Erste Schritte". Einige Details der Implementation verlassen sich darauf, dass die im Beispiel aufgeführten Initialisierungsschritte ausgeführt wurden.

Farben

Auf dem KC-85 sind nur 8 Hintergrundfarben (Farbcodes 0 bis 7) und 16 Vordergrundfarben (Farbcodes 0 bis 15 und 16 bis 31) möglich. Die Farbtöne von Vordergrund- und Hintergrundfarben unterscheiden sich, so dass insgesamt 24 verschieden Farben möglich sind. Werden die Farbcodes 16 bis 31 für die Vordergrundfarbe verwendet, so werden die entsprechenden Pixel blinkend dargestellt. Intern wird das folgendermaßen so realisiert:

- R-BASIC konvertiert die KC-Farbindizes intern in eine Farbe aus dem RGB-Würfel der GEOS Standardpalette, die den originalen KC-Farben möglichst nahe kommt. Eine entsprechende Zuordnung finden Sie im Anhang.
- Als Screen wird eine 8 Bit Bitmap mit Palette verwendet. R-BASIC stellt eine spezielle Palette ein, die diese Farben enthält. Das Blinken der Vordergrundfarben wird über einen Timer realisiert, der im Hintergrund etwa 2 Mal pro Sekunde die Palette der Screen-Bitmap modifiziert und die Bitmap neu dargestellt.

Variablen

Auf dem KC-85 werden Variablen bei ihrer ersten Verwendung definiert. Nur Felder müssen mit DIM angegeben werden. Unter R-BASIC müssen Sie alle Variablen zunächst mit DIM vereinbaren. Beachten Sie, dass auf dem KC nur die ersten beiden Buchstaben einer Variablen von Bedeutung sind! WE ist die gleiche Variable wie WELT und AL$ die gleiche wie ALF$.

IRM

Die Grafikausgabe des KC-85 erfolgt über einen Bildwiederholspeicher (IRM: Image Repeat Memory). Er besteht aus dem Pixelspeicher ab Adresse 0, dem Colorspeicher ab Adresse 10240, den Arbeitszellen des KC-Betriebssystems CAOS ab Adresse 12800 und einem freien Bereich ab Adresse 14848.
Im Pixelspeicher ist in jedem Byte für 8 Pixel die Information abgelegt, ob die Vordergrundfarbe oder die Hintergrundfarbe angezeigt werden soll. Der Colorspeicher enthält in jedem Byte für jeweils 4x8 Pixel die zugehörige Vordergrund- und die Hintergrundfarbe. Eine genaue Aufteilung des IRM sowie Informationen dazu, welche der CAOS Arbeitszellen von R-BASIC benutzt werden können, finden Sie im Anhang.
Aus BASIC heraus kann der IRM mit den Befehlen VPEEK und VPOKE (siehe unten) angesprochen werden. R-BASIC stellt dafür einen eigenen Speicherblock bereit. Je nachdem welche Bits in der Systemvariablen kc85Features gesetzt sind löst ein VPOKE Befehl weitere Aktionen aus, die das Verhalten des KC-85 nachahmen.
Sie sollten auf jeden Fall den zu portierenden Code nach Peek, Poke, VPeek, VPoke, Deek und Doke Befehlen durchsuchen um deren Funktion gegebenenfalls anpassen zu können.

Zeilennummern

Klassische BASIC Programme benötigen Zeilennummern, R-BASIC nicht. R-BASIC kann jedoch mit Zeilennummern umgehen, so dass Sie beim Portieren eines Programms die Zeilennummern nicht entfernen müssen.
Im Gegensatz zum KC-BASIC übernimmt R-BASIC reine Kommentarzeilen nicht in den Code. Folgende KC-Sequenz führt unter R-BASIC zu einem Fehler (Zeilennummer existiert nicht) wenn der Befehl "**GOSUB 1100**" compiliert wird.

```
1100  !  Unterprogramm zur Anzeige
1110  Window : COLOR 15, 1: CLS
```

Ändern Sie den Code dann folgendermaßen:

```
!  Unterprogramm zur Anzeige
1100  Window : COLOR 15, 1: CLS
```

Zeichensatz

Der Zeichensatz von KC und PC / R-BASIC unterscheiden sich an einige Stellen. Insbesondere müssen Sie die deutschen Umlaute über ihren Code in einen String einfügen, ein "ä" z.B. als "\125". Eine entsprechende Übersicht finden Sie im Anhang, Kapitel H.

Zeichengenerator

Einer der Initialisierungsschritte ist das Einstellen des BlockGrafik Modus. Wenn das Bit 9 in den kc85Features gesetzt ist kann R-BASIC erkennen, ob ein KC-85 Programm der Zeichengenerator des KC-85 ändert (Verwendung der Befehle POKE bzw. VPOKE) und seinen eigenen Zeichengenerator entsprechend anpassen. Wo der KC-Zeichengenerator liegt wird von den Arbeitszellen 14246 bis 14253 im IRM bestimmt, die mit dem Befehl VPOKE geändert werden können.

Details dazu finden Sie im Anhang. Der Standard-Zeichengenerator liegt beim KC im ROM auf den Adressen 60928 (&hEE00)bzw. 65024 (&hFE00). R-BASIC erkennt, wenn das Programm den Zeichengenerator in den RAM (Adresse kleiner 49152 bzw. &hC000) verschiebt und überwacht die entsprechenden Adressen dann. R-BASIC reagiert jedoch nicht adäquat, wenn das Programm die Adresse des Zeichengenerators wieder in den ROM zurück verschiebt. Das kann sinnvoll sein, wenn zwischenzeitlich wieder Buchstaben statt Grafikzeichen ausgegeben werden sollen. Sie müssen diese Programmstellen finden und ändern, indem Sie zwischenzeitlich auf den alternativen Zeichengenerator von R-BASIC wechseln. Im Normalfall müssen Sie dafür an zwei Stellen im Programm VPOKE durch BlockSelect ersetzen. Im eingangs genannten Beispiel ist das demonstriert.

Laufzeit

Läuft ein Programm zu schnell, z.B. in einer Spielschleife, so können Sie den Befehl DELAY verwenden. Wenn Sie während der zu schnell laufenden Schleife Strg-B drücken öffnet sich der R-BASIC Debugger und Sie können im Einzelschrittbetrieb die passende Position für die Delay-Befehle finden. Beachten Sie die Dokumentation zum Delay-Befehl!

Nicht vollständig KC-kompatible Befehle

Einige wenige Befehle sind nicht vollständig KC-kompatibel programmiert, um die Leistungsfähigkeit von R-BASIC nicht unnötig einzuschränken. In anderen Fällen würde der Nutzen im Vergleich zum erforderlichen Aufwand einfach zu gering.

Chr$: LEN(Chr$(0)) liefert auf dem KC den Wert 1, in R-BASIC Null.

InStr: Bei der Übergabe von Leerstrings gibt es Unterschiede. INTSR("","X") und INTSR("X","") erzeugen auf dem KC einen Laufzeitfehler, in R-BASIC erhalten wir Null.

INPUT: Bei der Eingabe von Strings werden führende Leerzeichen vom KC-85-BASIC ignoriert. R-BASIC übernimmt die Leerzeichen in den String.
Im KC-85-BASIC sind als Infotext nur Stringkonstanten zulässig, R-BASIC akzeptiert auch Variablen und String-Ausdrücke.

Left$(),
Right$(): Ist der Längenparameter negativ, erzeugt der KC einen Laufzeitfehler, R-BASIC liefert einen leeren String

Mid$(): Ist der Längenparameter negativ, erzeugt der KC einen Laufzeitfehler, R-BASIC liefert alle Zeichen ab dem Start-Parameter (so als ob kein Längenparameter angegeben wäre)

WINDOW:

Auf dem KC-85 hat der Grafikbildschirm eine feste Größe von 320 x 256 Pixeln. Da der Zeichensatz 8x8 Pixel groß ist, hat das Textfenster maximal 32 Zeilen zu je 40 Spalten. Unter R-BASIC kann der Bildschirm und somit das Textfenster eine beliebige Größe haben.

Der Befehl WINDOW (ohne Parameter) stellt auf dem KC-85 ein Standard-Fenster ein, dass oben und unten eine Zeile frei lässt. Das entspricht dem

Befehl WINDOW 1, 38, 0, 39. Unter R-BASIC stellt der Befehl WINDOW (ohne Parameter) das maximal mögliche Fenster ein.

Auf dem KC erzeugen Window-Parameter, die nicht zum Bildschirm passen (z.B. Anfangszeile < 0) einen Laufzeitfehler, in R-BASIC wird das Fenster angepasst (abgeschnitten) oder es wird das maximale Fenster eingestellt.

LOCATE: Der KC erzeugt einen Syntax-Fehler zur Laufzeit, wenn die Koordinaten ungültig sind. R-BASIC begrenzt die Koordinaten auf sinnvolle Werte und arbeitet weiter.

POS: Der Befehl wurde im CsrPos umbenannt, damit der Bezeichner pos für Variablen verwendet werden kann.

PRINT AT, INK, COLOR, PAPER : Auf dem KC in jeder Print-Anweisung nur ein AT und eine Farbanweisung zulässig. In R-BASIC kann man diese Anweisungen beliebig kombinieren.

PRINT - Steuercodes

In R-BASIC ist nicht definiert: &H14 (KeyKlick), &H16 (ShiftLock)
Folgende Codes arbeiten in R-BASIC nur innerhalb einer Zeile, während sie beim KC auch zeilenübergreifend arbeiten können: &H16 (Insert), &H17 (Delete) und &H08 (Clear Character)

WIDTH: Der Befehl wurde im KC_WIDTH umbenannt, damit der Bezeichner width für Variablen verwendet werden kann.

SOUND: Der Befehl wurde im KCSOUND umbenannt, damit der Bezeichner SOUND für einen leistungsfähigeren Sound-Befehl zur Verfügung steht. Die Länge eines "Dauertons" ist auf 18 Minuten begrenzt.

LINE, CIRCLE, PSET, PRESET: Die Grafikbefehle schreiben nicht in den IRM, weder in den Pixel- noch in den Color-RAM. Entsprechend kann es beim Lesen dieser Speicherbereiche mit VPEEK abweichende Ergebnisse geben.

VPOKE und VPEEK greifen auf einen von R-BASIC bereitgestellten Speicherblock zu. Über die Systemvariable kc85Fetaures (siehe unten) kann eingestellt werden, inwieweit R-BASIC versuchen soll, die damit verbundene Hardwareoperation des KC-85 nachzuahmen. In einigen Fällen gelingt das sehr gut, in anderen eher weniger gut.

INP, OUT, WAIT: Diese hardwarenahen Befehle greifen auf die I/O-Ports des PC zu. Die KC-Hardware wird nicht emuliert.

Nicht unterstützte KC-BASIC Befehle

Einige Befehle sind unter R-BASIC nicht sinnvoll (z.B. die Editorbefehle) oder konzeptionell nicht möglich (z.B. der Aufruf von Z80 Maschinencode).

- Editor-Befehle und Speicherverwaltung
  AUTO, BYE, CLOAD, CONT, CSAVE, DELETE, EDIT, FREE(), LINES, LIST, NEW, RENUMBER, RUN, TRON, TROFF
- Hardware- oder Maschinenprogramm-Zugriffe:
  JOYST, BLOAD, CALL, INPUT#, LIST#, LOAD#, SWITCH, WAIT, USR()

• Sonstige:
DEF_FN, KEY, KEYLIST, LET, FRE

• PTEST wird nicht unterstützt. Verwenden Sie stattdessen PGet.

Spezielle Befehle für die KC-Kompatibilität

kc85Features

Die Systemvariable kc85Features bestimmt das Kompatibilitätslevel von R-BASIC gegenüber dem KC-85 BASIC. Die in der Tabelle unten aufgelisteten Funktionen kann man einzeln ein- und ausschalten, da viele von ihnen R-BASIC einschränken oder verlangsamen. Dazu enthält die Variable kc85Features Bitflags, das heißt, jedes Bit hat eine eigene Bedeutung. Die Arbeit mit Bitflags ist im Kapitel 2.3.5.4 des Programmierhandbuchs beschrieben.
Die meisten Bits setzen voraus, dass der Ausgabescreen eine 8 Bit Bitmap mit Palette ist. Ist diese Bedingung nicht erfüllt funktionieren einige Funktionen nicht oder verhalten sich "seltsam".

Übersicht über die Bits in kc85Features

BitNr.	Wert	Aufgabe
0	1 (&h01)	Print ignoriert Koordinatentransformation
1	2 (&h02)	KC-85 Farben verwenden
2	4 (&h04)	Blinken unterstützen
3	8 (&h08)	VPOKE in den Pixel-RAM schreibt auch in die Ausgabebitmap
4	16 (&h10)	VPEEK aus dem Pixel-RAM liest aus der Ausgabebitmap
5	32 (&h20)	VPOKE in den Color-RAM schreibt auch in die Ausgabebitmap
6	64 (&h40)	VPEEK aus dem Color-RAM liest aus der Ausgabebitmap
7	128 (&h80)	VGet$ liest aus dem ASCII-Puffer im IRM
8	256 (&h100)	CLS löscht den Pixel-, Color-RAM sowie den ASCII-Puffer im IRM, PRINT und INPUT updaten den ASCII-Puffer im IRM.
9	512 (&h200)	KC-85 Zeichengenerator benutzen
10	1024 (&h400)	Erstbelegung der Buchstabentasten groß
11	2048 (&h800)	ASCII-Codes der Umlaute und Sonderzeichen von der Tastatur in KC-Codes umwandeln
12	4096 (&h1000)	INKEY$: ASCII-Codes der Steuertasten in KC-Codes umwandeln
13	8192 (&h2000)	Bit 3 der Arbeitszelle 14242 (&HB7A2) unterstützen
14	16344 (&h4000)	Peek(509) liest den Tastencode
15		Immer Null. Kann nicht geändert werden.

Nähere Beschreibung der einzelnen Bits

Bit 0 ermöglicht, dass das grafische Koordinatensystem seinen Ursprung links unten hat, während der PRINT Befehl weiterhin mit der Zeilen- und Spaltenzählung links oben beginnt, ohne dass die Buchstaben gespiegelt werden.

Bit 1 bewirkt, dass den KC-Farbcodes 0 bis 31 Farben aus der GEOS-Standardpalette zugeordnet werden, die denen des KC-85 entsprechen.

Bit 2 bewirkt, dass die Farben 16 bis 31 der Vordergrundfarbe blinkend dargestellt werden. Dazu wird etwa 2 Mal pro Sekunde die Palette der Screen-Bitmap modifiziert und die Bitmap neu dargestellt. Setzt voraus, dass das Bit 1 ebenfalls gesetzt ist. Falls Sie keine blinkenden Zeichen benötigen sollten Sie dieses Bit auch nicht setzen.

Bit 3 bewirkt, dass VPOKE beim Schreiben in den Pixel-RAM auch in die Ausgabebitmap schreibt. Die erzeugten Farben können aber vom Ergebnis auf dem KC abweichen.

Bit 4 bewirkt, dass VPEEK beim Lesen aus dem Pixel-RAM den zu lesenden Wert aus den Pixeldaten der Ausgabebitmap rekonstruiert. Ist das Bit nicht gesetzt wird der Wert direkt aus dem IRM gelesen.

Bit 5 bewirkt, dass VPOKE beim Schreiben in den Color-RAM auch in die Ausgabebitmap schreibt.

Bit 6 bewirkt, dass VPEEK beim Lesen aus dem Color-RAM den zu lesenden Wert aus den Pixeldaten der Ausgabebitmap rekonstruiert. Der gelesene Wert kann vom Ergebnis auf dem KC abweichen. Ist das Bit nicht gesetzt wird der Wert direkt aus dem IRM gelesen.

Bit 7 bewirkt, dass VGet$ den Wert aus dem ASCII-Puffer im IRM liest statt aus dem R-BASIC ASCII-Puffer. Setzt einen 40 Zeilen x 32 Zeichen - Screen voraus.

Bit 8 bewirkt, dass PRINT, INPUT und CLS den ASCII-Puffer im IRM updaten. CLS löscht außerdem den Pixel- und den Color-RAM. Setzt einen 40 Zeilen x 32 Zeichen - Screen voraus.

Bit 9 bewirkt, dass der KC85 Zeichengenerator in den R-BASIC Zeichengenerator gespiegelt und dabei gegebenenfalls skaliert wird, indem die entsprechenden Arbeitszellen im IRM überwacht werden. Setzt einen aktiven Blockgrafik-Modus mit einem 40 Zeilen x 32 Zeichen - Screen voraus. Die Blockgrafikzeichen müssen quadratisch und ihre Größe durch 8 teilbar sein. Ändert das KC-Programm den Zeichengenerator im RAM so passt R-BASIC seinen eigenen Zeichengenerator automatisch an.

Bit 10 bewirkt, dass die Erstbelegung der Buchstabentasten wie beim KC die Großbuchstaben sind.

Bit 11 bewirkt, dass die ASCII-Codes der Umlaute und Sonderzeichen auf der Tastatur für INPUT und INKEY$ in die entsprechenden KC-Codes umgewandelt werden. Hinweis: Auf dem KC gibt es bei den Umlauten keine Großbuchstaben.

Bit 12 bewirkt, dass INKEY$ bei den Steuertasten (Pfeiltasten, Einfg usw.) die KC-Codes liefert. INPUT behandelt die Steuertasten immer automatisch korrekt.

Bit 13 bewirkt, dass das Bit 3 der Arbeitszelle 14242 (&HB7A2) im IRM kontrolliert, ob Steuerzeichen (ASCII-Code < 32) "ausgeführt" oder als druckbare Zeichen ausgegeben werden. Dazu wird das Bit TS_DONT_EXEC_ CONTROLS in printFont.style angepasst.

Bit 14 bewirkt, dass der Befehl Peek(509) des Tastaturcode der aktuell gedrückten Taste liefert. Die Speicherstelle mit der Adresse 509 (=&h1FD) ist eine Arbeitszelle des CAOS Betriebssystem des KC-85.

Hinweis: Die R-BASIC Befehle GetKey bzw. GetKeyLP liefern nicht den KC-Code sondern weiterhin den R-BASIC Code. Diese Codes unterscheiden sich für Steuerzeichen. Will man sie verwenden muss man möglicherweise die Abfrage entsprechend anpassen.

Bit 15 ist immer Null und kann nicht geändert werden.

VPOKE

VPOKE schreibt ein Byte in den IRM. In diesem Bereich liegt der Bildwiederholspeicher des KC-85 und die Arbeitszellen des KC Betriebssystems. Einige Bits in der Systemvariablen kc85Fetaures bestimmen, ob VPOKE weitere Aktionen auslöst oder einfach nur den Speicher beschreibt.
Eine genaue Aufteilung des IRM sowie Informationen dazu, welche der CAOS Arbeitszellen von R-BASIC benutzt werden können, finden Sie im Anhang.

Syntax: **VPOKE adr**, **val**
adr: Adresse im IRM. Erlaube Werte sind 0 bis 16383
val: Zu schreibender Wert (Byte).

VPEEK

VPEEK liest ein Byte aus dem IRM. Die Bits 4 und 6 der Systemvariablen kc85Features bestimmen, ob beim Lesen aus dem Pixel- bzw. Color-RAM der IRM selbst gelesen werden soll oder ob VPEEK versuchen soll, den Wert aus den Pixeldaten der Ausgabebitmap zu rekonstruieren.

Syntax: **<numVar>** = **VPEEK adr**
adr: Adresse im IRM. Erlaube Werte sind 0 bis 16383

PEEK(509)

Die Speicherstelle mit der Adresse 509 (=&h1FD) ist eine Arbeitszelle des CAOS Betriebssystem des KC-85. Dort wird der Code der aktuell gedrückten Taste abgelegt. Viele KC-BASIC Programme greifen direkt darauf zu. Das Setzen des Bits 14 in der Systemvariablen kc85Features bewirkt, dass der Befehl Peek(509) des Tastaturcode der aktuell gedrückten Taste liefert.

POKE 862, 1

Die Speicherstelle mit der Adresse 862 (=&h35E) ist eine Arbeitszelle des BASIC Interpreters des KC-85. Ein Wert ungleich Null verhindert, dass der Nutzer den Quellcode eines BASIC Programms ansehen kann (Listschutz). R-BASIC ignoriert diese Zelle.

KC_WIDTH

KC_WIDTH (Breite, KC Kompatibilitätsbefehl) bestimmt die maximale Länge einer Ausgabezeile. Dieser Befehl ist ursprünglich zur Arbeit mit Druckern gedacht und wurde deswegen umbenannt. Der Originalbefehl im KC-85 BASIC lautet WIDTH.

Syntax:	**KC_WIDTH n**
n:	Länge der Ausgabezeile. Werden mehr als n Zeichen auf einmal ausgegeben, wird nach n Zeichen jeweils ein Zeilenumbruch eingefügt.

Beispiel:

```
KC_WIDTH 15
```

KCClear

KCCLEAR löscht den globalen Variablenspeicher, einschließlich String und HUGE Variablen. Der Originalbefehl im KC-85 BASIC lautet CLEAR.

Syntax: **KCClear**

Beispiel:

```
KCClear
```

19 Objekte individualisieren

Es gibt Situationen, in denen es nötig ist, dass ein Objekt neben den vom System vorgegebenen Daten weitere Informationen speichern muss. Ein einfaches Beispiel ist ein Canvas-Objekt, das entweder einen Kreis oder ein Quadrat zeichnen soll. Sie können die Information, ob ein Kreis oder ein Quadrat gezeichnet werden soll, natürlich in einer globalen Variablen speichern. Das ist aber nicht nur schlechter Stil sondern wird bei mehreren solchen Objekte auch schnell sehr unübersichtlich und damit fehlernfällig.
Die bessere Lösung ist, die Information im Objekt selber zu speichern. R-BASIC bietet Ihnen für diese Situation die Instancevariable **privData**. PrivData nimmt eine Strukturvariable beliebigen Typs auf und speichert sie im Objekt selbst. Hier können Sie z.B. ablegen, ob ein Kreis oder ein Quadrat gezeichnet werden soll. Außerdem können Sie - wenn Sie wollen - die Größe, die Farbe und beliebige weitere Informationen speichern.
Für einfache Fälle steht Ihnen bei Objekten der Klasse **VisObj** zusätzlich die Instancevariable **visDataValue** zur Verfügung, die einen numerischen Wert aufnehmen kann und beim VisObj-Objekt beschrieben ist.

Fortgeschrittene Programmierer können in einigen Situationen den Bedarf haben, dass sie eine Routine erst dann aufrufen wollen, wenn der aktuelle Actionhandler vollständig abgearbeitet ist. Typische Beispiele sind hier der OnPrint Handler (bei dem man das Screen-Objekt nicht ändern darf) oder ein OnMouse~ bzw. der OnKeyPressed Handler (die meist zeitkritisch sind). R-BASIC löst dieses Problem, indem man für Objekte eigene, private ("custom") Handler definieren kann. Actionhandler unterbrechen sich niemals, sondern werden immer nacheinander abgearbeitet. Der Aufruf eines solchen Handlers führt also dazu, dass die aktuelle Routine (genauer: der komplette aktuell laufende Handler) zuerst vollständig abgearbeitet wird bevor der neue Handler ausgeführt wird. Um einen Custom Handler für ein Objekt festzulegen verwenden Sie die Instancevariable **CustomHandler**. Custom Handler müssen als **CustomAction** deklariert sein. Um einen Custom Handler aufzurufen verwenden Sie die Methode **CustomApply**.

PrivData

PrivData ist für alle Klassen definiert. Sie nimmt eine einzelne Strukturvariable (also maximal 3500 Bytes) auf. Diese Instancevariable ist zuweisungskompatibel mit jeder Art von Struktur, es wird weder eine Typ- noch eine Größenprüfung ausgeführt. Es ist daher vernünftig beim Schreiben und beim Lesen der Daten den gleichen Struktur-Datentyp zu verwenden.

Schreiben: Sie müssen die Größe der zu schreibenden Daten angeben. Verwenden Sie dazu die Funktion SIZEOF.

Lesen: Es werden so viele Bytes gelesen, wie die Variable auf der linken Seite der Zuweisung aufnehmen kann. Enthält privData weniger Bytes, so wird der Rest mit Nullen aufgefüllt.

Es ist zulässig mehrfach hintereinander Strukturen verschiedenen Typs und verschiedener Größe in die Instancevariable zu schreiben. R-BASIC optimiert jedes Mal den verwendeten Speicher, so dass kein Platz verschwendet wird.

Syntax Schreiben: **<obj>.privData** = **<struct>**, **size**
Lesen: **<structVar>** = **<obj>.privData**
<struct>: Strukturausdruck beliebigen Typs
size Größe der Struktur
<structVar>: Strukturvariable des Typs, der beim Schreiben verwendet wurde.

Achtung! PrivData kann nicht im UI Code belegt werden! Wenn Sie wollen, das privData am Programmstart mit vorgegebenen Werten belegt wird, müssen Sie das im OnStartup- oder im OnInit-Handler tun.

Beispiel:
Ein Canvas-Objekt soll einen Kreis oder ein Quadrat in einer vorgegebenen Farbe zeichnen. Wir benötigen:
- einen Strukturtyp, der die Informationen enthält,
- eine Routine, die die Werte setzt,
- ein Canvas-Objekt,
- einen OnDraw Handler für das Canvas Objekt,
- einen OnStartup Handler für das Application Objekt

Der Strukturtyp sei folgendermaßen definiert:

```
STRUCT ImgData
  isCircle   as Integer
  color      as Integer
End Struct
```

Zum Belegen der Instancewerte dient die folgende Routine. Die zweite Routine (SetCanvasToRect) ist hier nicht aufgeführt.

```
SUB SetCanvasToCircle( col as Integer)
DIM pd AS ImgData
  pd.isCircle = TRUE
  pd.color = col
  DemoCanvas.privData = pd, SIZEOF(pd)
  DemoCanvas.Dirty                    ' Neudarstellung auslösen
End Sub
```

Das Canvas-Objekt sei wie folgt definiert. Beachten Sie, dass wir privData nicht definieren brauchen, es ist für alle Objekte automatisch verfügbar.

```
CANVAS  DemoCanvas
  OnDraw = DrawFigure
  fixedSize = 70, 70
End Object
```

Unser Application-Objekt benötigt einen OnStartup Handler.

```
Application DemoApplication
  Children = DemoPrimary
  OnStartup = AppStartup
End OBJECT
```

Schließlich benötigen wir noch den OnDraw-Handler, der die privData-Werte ausliest und verwendet sowie den OnStartup Handler für das Application-Objekt.

```
DRAWACTION DrawFigure
DIM priv as ImgData

  priv = sender.privData
  INK priv.color
  IF priv.isCircle THEN
    FillEllipse 10, 10, 60, 60
  ELSE
    FillRect 10, 10, 60, 60
  END IF
End Action
```

```
SYSTEMACTION AppStartup
  SetCanvasToCircle(LIGHT_RED)
END ACTION
```

CustomHandler

CustomHandler enthält den Namen des Actionhandlers, der mit der Methode CustomApply aufgerufen werden soll.

Syntax UI- Code: **CustomHandler** = **<Handler>**
Schreiben: **<obj>**.**CustomHandler** = **<Handler>**

Ein Custom Handler muss als CustomAction deklariert sein:

Handler-Typ	Parameter
CustomAction	(sender as object, actionData as integer)

CustomApply

Die Methode CustomApply ruft den CustomHandler eines Objekts auf. Ihr wird ein Integer-Wert übergeben, der an den Handler weitergereicht wird.

Syntax: **<obj>**.**CustomApply** **actionData**
actionData: Integerwert, der an CustomHandler übergeben wird.

Beispiel (einfach, deswegen nicht sehr sinnvoll):

Ein Button mit einem ActionHandler und Primary mit einem CustomHandler.

```
BUTTON MyButton
  Caption$ "Drück mich!"
  ActionHandler = PressHandler
End Object

BUTTON MyPrimary
  CustomHandler = CHandler
End Object
```

Der Actionhandler:

```
ButtonAction PressHandler
  Print "Text 1"
  MyPrimary.CustomApply   1
  Print "Text 2"
  MyPrimary.CustomApply   -7
  Print "Text 3"
End Action
```

Der CustomHandler wird erst ausgeführt, wenn der ActionHandler fertig ist

```
CustomAction CHandler
DIM i
  IF actionData < 0 THEN i = RED : ELSE i = BLUE
  Print Ink i;"DATA = ";actionData
End Action
```

Wenn der Nutzer den Button drückt erscheint folgendes:

```
  Text 1
  Text 2
  Text 3
  DATA = 1
  DATA =-7
```