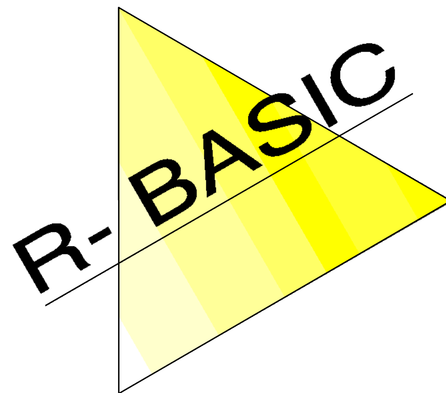
R-BASIC

Einfach unter PC/GEOS programmieren



Objekt-Handbuch

Volume 4
Dialog, Number

Version 1.0

(Leerseite)

Inhaltsverzeichnis

4.6 Dialog 152
4.6.1 Überblick 152
4.6.2 Allgemeine Eigenschaften 154
4.6.3 Öffnen und Schließen von Dialogboxen 156
4.6.4 Behandlung von Messages 159
4.6.4.1 Messages von UI-Objekten 159
4.6.4.2 InteractionCommand 159
4.6.4.3 Behandlung von Dialog-Messages 162
4.6.5 Frei definierte Dialoge 165
4.6.6 Standard-Dialoge 167
4.6.6.1 Command-Dialoge 168
4.6.6.2 Notification-Dialoge 168
4.6.6.3 Question-Dialoge 169
4.6.6.4 Progress-Dialoge 169
4.6.6.5 Dialoge im Delayed Mode 172
4.6.6.6 Eigene Buttons in Standard-Dialogen 174
4.6.7 Arbeit mit Blocking-Dialogen 176

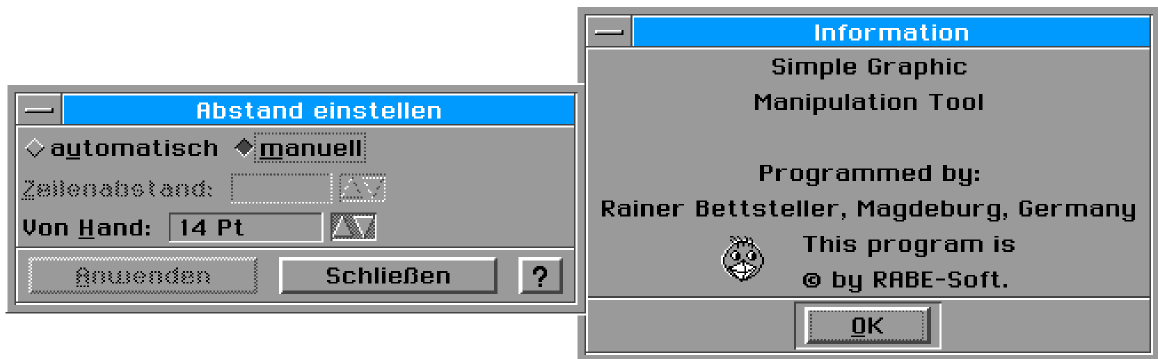
4.7 Number 180
4.7.1 Grundlegende Eigenschaften 181
4.7.2 Display-Format 185
4.7.3 Angepasstes Aussehen und Sliders 188
4.7.4 Weitere Hinweise zur Arbeit mit Number-Objekten 191
4.7.5 Number-Objekte im Delayed Mode 193

(Leerseite)

4.6 Dialog

4.6.1 Überblick

Dialoge sind unabhängige Fenster, mit denen der Nutzer interagieren kann. Sie werden für die unterschiedlichsten Aufgaben eingesetzt (siehe Bilder). Deshalb verfügt das Dialog-Objekt über sehr viele Fähigkeiten, die in den folgenden Abschnitten beschrieben werden.



Focus und Target

Dialog-Objekte sind ein Knoten in der Focus- und Target-Hierarchie. Es ist möglich zu überwachen, ob ein Dialog-Objekt den Focus oder das Target hat, indem man einen Focus- bzw. Target-Handler schreibt. Die notwendigen Details zur Arbeit mit Focus und Target finden Sie im Kapitel 12 (Focus und Target) des Handbuchs "Spezielle Themen". Das Arbeiten mit Focus und Target ist etwas für erfahrene Programmierer und nur in wenigen Fällen notwendig. Eine Ausnahme bildet die Implementation von speziellen Menüs wie dem "Bearbeiten" Menü. Diesem Thema ist deswegen ein eigenes Kapitel ("Spezielle Themen", Kapitel 13) gewidmet.

Abstammung

Die Möglichkeiten, einen Dialog einzusetzen sind sehr vielfältig. In diesem Zusammenhang sollten Sie die folgenden Begriffe kennen:

Frei definierte Dialoge

Als "frei definierte" Dialoge werden in diesem Handbuch Dialogboxen bezeichnet, deren Objekte Sie vollständig selbst definieren. Frei definierte Dialoge sind für Einsteiger gut überschaubar und werden im Kapitel 4.6.5 besprochen.

Standard-Dialoge

Standard-Dialoge enthalten bereits einige vordefinierte Objekte, z.B. Buttons. Dadurch wird dem Programmierer viel Arbeit abgenommen, aber er muss sich im Gegenzug mit den Standard-Dialogtypen und der Thematik der interactionCommand-Werte auseinandersetzen.

"Blocking" Dialoge

Blocking-Dialoge zeichnen sich dadurch aus, dass sie das laufende Programm warten lassen (blockieren), bis der Nutzer mit der Arbeit mit dem Dialog fertig ist. Sie werden für wichtige Informationen (z.B. Fehlermeldungen) oder Nachfragen benutzt. Blocking-Dialoge können sowohl frei definierte als auch Standard-Dialoge sein.

Reply-Bar

Eine Reply-Bar (so viel wie Reaktionsleiste) ist eine Group am unteren Rand der Dialogbox, in der sich meist die Reaktions-Buttons eines Dialogs befinden. Die Dialoge im Bild oben haben jeweils eine Reply-Bar mit drei bzw. einem Button. Reply-Bars ordnen ihre Children auf spezielle Weise an. Standard-Dialoge haben automatisch immer eine Reply-Bar, Sie können mit dem Hint MakeReplyBar aber auch eine eigene Group zu einer Reply-Bar machen.

Im Folgenden finden Sie eine vollständige Liste der Instance-Variablen, Methoden, Handler-Typen und Routinen eines Dialog-Objekts. Eine Beschreibung dieser finden Sie in den folgenden Kapiteln.

Spezielle Instance-Variablen:

Variable	Syntax im UI-Code	Im BASIC-Code
MakeResizable	MakeResizable	—
NoFocus	NoFocus	—
attrs	attrs = **numWert**	lesen, schreiben
modal	modal = **numWert**	lesen, schreiben
isOpen	—	nur lesen
dialogType	dialogType = **numWert**	lesen, schreiben
interactionCommand	—	nur lesen
OnOpen	OnOpen = **<Handler>**	nur schreiben
OnClose	OnClose = **<Handler>**	nur schreiben
OnCommand	OnCommand = **<Handler>**	nur schreiben

Methoden:

Methode	Aufgabe
Open	Dialog auf den Schirm bringen (öffnen)
OpenNoDisturb	Dialog ohne Übernahme des Focus öffnen
Close	Dialog schließen

Action-Handler-Typen:

Handler-Typ	Parameter
DialogAction	(sender as object, command as integer)

Spezielle Routinen: **OpenBlockingDialog**(dialogObj as object)

4.6.2 Allgemeine Eigenschaften

Variable	Syntax im UI-Code	Im BASIC-Code
MakeResizable	MakeResizable	—
NoFocus	NoFocus	—
attrs	attrs = **numWert**	lesen, schreiben
modal	modal = **numWert**	lesen, schreiben
isOpen	—	nur lesen

MakeResizable

Normalerweise sind Dialogboxen nicht größenveränderlich. Der Hint MakeResizable bewirkt, dass man die Größe des Dialogs auf dem Bildschirm verändern kann.

Syntax UI-Code: **MakeResizable**

NoFocus

Normalerweise übernimmt ein Dialog, wenn er geöffnet wird, oder der Nutzer mit der Maus ein Objekt im Dialog anklickt, automatisch den Focus, d.h. alle folgenden Tastatureingaben gehen an den Dialog. In Situationen, in denen dieses Verhalten störend ist, können Sie mit dem Hint NoFocus verhindern, dass der Dialog Tastatureingaben entgegennimmt. Wenn der Nutzer z.B. gerade einen Text eingibt, kann ein solcher Dialog mit der Maus bedient werden, ohne dass der Text den Focus verliert.

Syntax UI-Code: **NoFocus**

attrs

Die Instance-Variable **attrs** speichert Attribute (spezielle Eigenschaften) des Dialogs. Dabei stehen die in der Tabelle aufgeführten Werte zur Verfügung. Sie werden weiter unten, an passender Stelle, ausführlich beschrieben.

Konstante	Wert	Bedeutung
	0	Keine Besonderheiten
DA_HIDDEN_UNTIL_OPENED	1	Öffnen nur durch das Programm, nicht durch den Nutzer direkt
DA_BLOCKING	2	Programmausführung wird angehalten, bis Dialog beendet ist

Syntax UI-Code: **attrs** = **numWert**
Lesen: **<numVar>** = **<obj>.attrs**
Schreiben: **<obj>.attrs** = **numWert**
numWert ist eine der DA_-Konstanten oder Null

modal

Konstante	Wert	Bedeutung
NON_MODAL	0	Nicht-modaler Dialog
APP_MODAL	1	Application-modaler Dialog
SYS_MODAL	2	System-modaler Dialog

Der Begriff "Modalität", beschreibt, inwieweit Eingaben (Tastatur und Maus) exklusiv an die Dialogbox gehen sollen. Bei einem nicht-modalen Dialog (NON_MODAL) können Sie beliebig zwischen der Dialogbox und dem Rest der Applikation oder des Systems hin- und her wechseln. Ein Beispiel wäre der Dialog "Linienattribute" aus GeoDraw. Ein Application-modaler Dialog (APP_MODAL) blockiert den Rest der Applikation, andere Anwendungen lasen sich weiter bedienen. Beispielsweise ist der "Speichern unter" Dialog von GeoWrite Application-modal. System-modale Dialoge (SYS_MODAL) blockieren die Bedienung des gesamten restlichen GEOS-Systems. Ein Beispiel ist die in bestimmten Situationen von GEOS erzeugte Nachfrage, ob das System heruntergefahren werden soll.

Syntax UI-Code: **modal** = **numWert**
Lesen: **<numVar>** = **<obj>.modal**
Schreiben: **<obj>.modal** = **numWert**
numWert ist eine der ~_MODAL-Konstanten von oben

isOpen

Diese nur-Lesen Variable enthält die Information, ob der Dialog aktuell offen ist (auf dem Schirm sichtbar, isOpen = TRUE) oder nicht (isOpen = FALSE).

Syntax Lesen: **<numVar>** = **<obj>.isOpen**

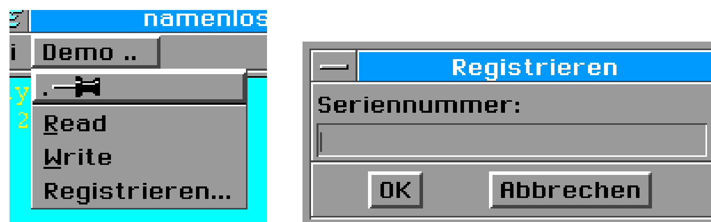
4.6.3 Öffnen und Schließen von Dialogboxen

Damit ein Dialog auf dem Bildschirm erscheinen kann (**geöffnet** werden kann), muss er **grundsätzlich** in den generic Tree des Programms eingebunden sein. Es ist ein häufiger Fehler, dies bei Dialogen, nicht nach Methode 1 (siehe unten) verwendet werden, zu vergessen. R-BASIC verfügt prinzipiell über 3 Methoden, einen Dialog zu öffnen.

Methode 1

Die einfachste Methode, einen Dialog zu verwenden, ist ihn als Child eines Menüs in den generischen Tree einzubinden. Das GEOS-System erzeugt dann automatisch einen Button im Menü, der den Dialog öffnet.

Beispiel:



UI-Code-Fragment. Ausführliche Code-Beispiele finden Sie im Kapitel 4.6.3.

```
Menu DemoMenu
  Caption$ = "Demo .. "
  Children = Readbutton, Writebutton, RegisterDialog
  END Object

Dialog RegisterDialog
  Caption$ = "Registrieren"
  Children = SerialText, OKButton, CancelButton
  END Object
```

Tipp: Wenn es zeitweise keinen Sinn macht, dass der Nutzer den Dialog öffnen kann, setzen Sie ihn zwischenzeitlich auf "not enabled" (**<dialogObj>**.enabled = FALSE).

Methode 2

Syntax im BASIC-Code	Aufgabe
<dialogObj>.**Open**	Dialog auf den Schirm bringen (öffnen)
<dialogObj>.**OpenNoDisturb**	Dialog ohne Übernahme des Focus öffnen
<dialogObj>.**Close**	Dialog schließen

attrs = DA_HIDDEN_UNTIL_OPENED	Nicht durch Nutzer zu öffnen

Die nach Methode 1 eingebundenen Dialoge kann der Nutzer prinzipiell jederzeit öffnen. Oftmals ist es aber nötig, Dialoge vom Programm aus zu öffnen, wenn es die Situation erfordert. Für diesen Zweck verfügten Dialog-Objekte über Methoden zum Öffnen und Schließen der Dialogbox.

Open Die Dialogbox wird geöffnet. Der Focus geht an die Dialogbox (es sei denn, sie hat den Hint NoFocus gesetzt), d.h. der Dialog ist das "aktive" Fenster und nimmt ab sofort alle Tastatur-Eingaben entgegen.

OpenNoDisturb (Öffnen ohne zu stören). Der Dialog wird geöffnet, bekommt aber noch nicht den Focus. Der Nutzer wird bei seiner aktuellen Arbeit am Programm (z.B. einen Text einzugeben) nicht gestört. Wenn er den Dialog bedienen will, klickt er einfach auf den Dialog. Der Dialog bekommt dann den Focus, d.h. er kann mit Maus und Tastatur bedient werden.

Der Unterschied zwischen einem Dialog, der mit **OpenNoDisturb** geöffnet wurde und einem, der den Hint **NoFocus** gesetzt hat ist folgender:
NoFocus-Dialoge bekommen den Focus auch dann nicht, wenn sie mit der Maus angeklickt werden. Eine Bedienung mit der Tastatur oder eine Texteingabe ist bei NoFocus-Dialogen nicht möglich.

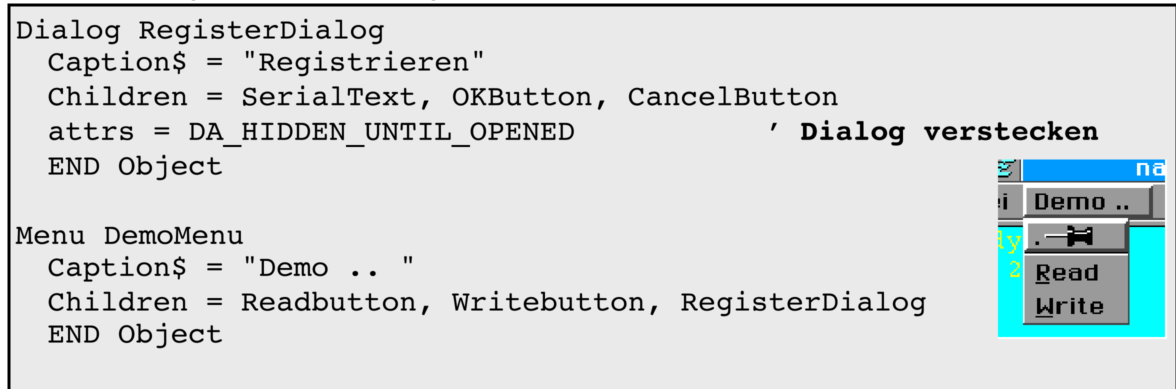
Close Der Dialog wird geschlossen. Weitere Möglichkeiten, einen Dialog zu schließen, werden in den nächsten Kapiteln behandelt.

Bei Dialogen, die mit **Open** oder **OpenNoDisturb** geöffnet werden, ist es oft gar nicht sinnvoll, ihn auch über ein Menü öffnen zu können. Trotzdem **müssen** solche Dialoge in den generischen Tree eingebunden werden. Um zu verhindern, dass das GEOS-System einen Aktivierungs-Button für diesen Dialog anlegt, verwenden Sie im UI-Code die Zeile

```
attrs = DA_HIDDEN_UNTIL_OPENED
```

Der Dialog wird solange vor dem Nutzer versteckt (engl. hidden = versteckt) bis er explizit durch eine der Anweisungen **Open** oder **OpenNoDisturb** geöffnet wird (until = bis, opened = geöffnet) .

UI-Code-Fragment. Der Dialog erscheint nicht im Menü!




```
Dialog RegisterDialog
  Caption$ = "Registrieren"
  Children = SerialText, OKButton, CancelButton
  attrs = DA_HIDDEN_UNTIL_OPENED            ' Dialog verstecken
  END Object

Menu DemoMenu
  Caption$ = "Demo .. "
  Children = Readbutton, Writebutton, RegisterDialog
  END Object
```

BASIC-Code-Fragment

```
  IF registerNr <> validNr THEN   RegsiterDialog.Open
```

Beachten Sie, dass R-BASIC nach dem Öffnen des Dialogs sofort mit der Abarbeitung der nächsten Codezeilen fortsetzt. Es wird nicht gewartet bis der Dialog wieder geschlossen wird. Ist das nicht gewollt, verwenden Sie bitte Methode 3.

Methode 3

Syntax im BASIC-Code	Aufgabe
numVar = **OpenBlockingDialog**(<dialogObj>)	Dialog aktivieren

attrs = DA_BLOCKING	Auf Reaktion durch Nutzer warten

Oftmals ist es für den Programmablauf erforderlich, dass der Nutzer zuerst den Dialog bedient, bevor die Programmabarbeitung fortgesetzt werden kann. Ein Beispiel wäre die Nachfrage, ob die Daten gespeichert werden sollen oder nicht.
Für diesen Zweck gibt es die Funktion **OpenBlockingDialog()**, die auf eine Eingabe des Nutzers wartet und solange die weitere Programmabarbeitung blockiert ("Blocking"). Sie liefert einen numerischen Wert zurück, je nachdem, welchen Button des Dialogs der Nutzer gedrückt hat. Solche Dialoge **müssen**

```
  attrs = DA_BLOCKING
```

gesetzt haben. DA_BLOCKING impliziert DA_HIDDEN_UNTIL_OPENED, d.h. das System erzeugt keinen "Aktivierung-Button". Vergessen Sie aber nicht, den Dialog an irgendeiner Stelle in den generic Tree einzubinden.
Außerdem ist es erforderlich, dass ein Blocking-Dialog modal ist (APP_MODAL oder SYS_MODAL). Geben Sie keinen Wert vor, setzt R-BASIC automatisch modal = APP_MODAL.
Eine ausführliche Beschreibung der Arbeit mit Blocking-Dialogen und passende Code-Beispiele finden Sie im Kapitel 4.6.7

4.6.4 Behandlung von Messages

4.6.4.1 Messages von UI-Objekten

UI-Objekte, die sich in einem Dialog befinden (Buttons, Listen, InputLine-Texte...) können grundsätzlich Messages versenden (d.h. ihre Action-Handler aufrufen). Ebenso kann man mit diesen Objekten arbeiten (z.B. Instance-Variablen belegen oder abfragen), wenn der Dialog nicht offen (nicht auf dem Schirm) ist.

Eine Ausnahme gibt es bei Blocking-Dialogen. Objekte in Blocking-Dialogen dürfen keine Action-Handler haben. Details dazu finden Sie im Kapitel 4.6.7.

4.6.4.2 InteractionCommand

Variable	Syntax im UI-Code	Im BASIC-Code
interactionCommand	—	nur lesen

Die Instance-Variable **interactionCommand** dient der direkten Kommunikation zwischen Button-Objekten in einen Dialog und dem Dialog-Objekt selbst, ohne dass Sie als R-BASIC Programmierer eingreifen müssen. Dies wird für Standard-Dialoge und für Blocking-Dialoge benötigt. Ein **interactionCommand**-Wert ist eine Zahl (Datentyp WORD). Er wird vom Button an sein Dialog-Objekt gesendet, wenn er angeklickt wird. Dadurch kann der Dialog bestimmte Aktionen automatisch ausführen, z.B. sich selbst schließen oder einen seiner Action-Handler aufrufen (**OnOpen**, **OnClose** oder **OnCommand**, siehe Kapitel 4.6.4.3).

Zu diesem Zweck besitzen sowohl Buttons als auch Dialoge eine Instance-Variable namens **interactionCommand**. Den **interactionCommand**-Wert eines Buttons kann man lesen und schreiben, üblicher Weise wird er im UI-Code gesetzt. Den **interactionCommand**-Wert eines Dialogs kann man nur lesen. Er wird vom Dialog automatisch belegt. Beim Öffnen des Dialogs wird der Wert auf Null gesetzt.

Syntax Lesen **numVar** = **<dialogObj>.interactionCommand**

Die folgenden Werte sind vom System definiert. Sie können auch eigene Werte definieren. Eigene **interactionCommand**-Werte müssen größer als 999 sein, die Werte 0 bis 999 sind vom System reserviert!

interactionCommand Konstante	Wert	Bedeutung
IC_CLOSE	1	Dialog schließen
IC_APPLY	3	Änderungen anwenden
IC_RESET	4	Dialog zurücksetzen
IC_OK	5	Verwendet für "OK"
IC_YES	6	Verwendet für "Ja"
IC_NO	7	Verwendet für "Nein"
IC_STOP	8	Verwendet für "Stopp" oder "Abbrechen"
IC_HELP	10	Button ersetzt den "Hilfe" Button

Intern passiert folgendes:

Nehmen wir an, wir haben einen Button in einem Dialog-Objekt, dessen **interactionCommand**-Wert belegt ist. Wird dieser Button angeklickt, so sendet er seinen **interactionCommand**-Wert direkt an den Dialog. Der Dialog reagiert darauf in Abhängigkeit vom **interactionCommand**-Wert:

IC_CLOSE:

- Die Dialogbox wird geschlossen.
- Die Instance-Variable **interactionCommand** des Dialogs wird nicht verändert, es sei denn, sie ist noch Null, dann wird sie mit IC_CLOSE (= 1) belegt.
- Falls vorhanden wird der **OnClose** Handler aufgerufen

IC_HELP:

- Wenn eine Dialogbox einen Wert für **helpContext$** gesetzt hat erzeugt sie automatisch einen "Hilfe" Button, dessen Beschriftung ein Fragezeichen ist. Um diese Aufschrift zu ändern muss man einen Button anlegen, dessen **interactionCommand** Wert auf IC_HELP gesetzt ist. Dieser Button ersetzt dann den vom Dialog erzeugten Button. Zusätzlich sollten Sie dem Button die Anweisung
  placeObject = REPLY_BAR
  geben, falls das angebracht ist.
  Das Dialogobjekt oder eines seiner Parents, üblicherweise das Application Objekt, sollte einen Wert für **helpFile$** gesetzt haben.

```
Dialog HelpedDialog
  caption$ = "Dialog mit Hilfe"
  children = ... , DilaogHelpButton
  dialogtype = DT_COMMAND
  helpContext$="MoreHelp"
  End Object
...
Button DilaogHelpButton
  Caption$ = "Hilf mir"
  interactionCommand = IC_HELP
  placeObject = REPLY_BAR
  End Object
```

- Falls der Dialog keinen Help Context gesetzt hat fordert der Button beim Application Objekt den Namen der Hilfedatei und einem Help Context an.
- Sollte die Hilfedatei oder der Help Context in der Hilfedatei nicht gefunden werden erzeugt das Hilfesystem eine Fehlermeldung.
- Abschließend wird, falls vorhanden, der **OnCommand** Handler aufgerufen.

IC_xxx: (alle sonstigen Werte)

- Die Instance-Variable **interactionCommand** des Dialogs wird mit dem vom Button kommenden Wert belegt.
- Falls der Button ein Standard-Button eines Standard-Dialogs ist (siehe Standard-Dialoge, Kapitel 4.6.6) wird der Dialog geschlossen und, falls vorhanden, der OnClose Handler aufgerufen.
- Zuletzt wird, falls vorhanden, der **OnCommand** Handler aufgerufen.

Ein Zugriff auf die Instance-Variable **interactionCommand** des Dialogs ist nur selten nötig, insbesondere aber dann, wenn der Dialog das Abbrechen von länger dauernden Prozessen ermöglichen soll (siehe Progress-Dialoge, Kapitel 4.6.6.4).

4.6.4.3 Behandlung von Dialog-Messages

Instance-Variable	Syntax im UI-Code	Im BASIC-Code
OnOpen	OnOpen = **<Handler>**	nur schreiben
OnClose	OnClose = **<Handler>**	nur schreiben
OnCommand	OnCommand = **<Handler>**	nur schreiben

Action-Handler-Typen:

Handler-Typ	Parameter
DialogAction	(sender as object, command as integer)

Aus Sicht einer Dialogbox gibt es drei wichtige Ereignisse:
1. Die Dialogbox wird geöffnet
2. Die Dialogbox wird geschlossen
3. Ein Button mit einem interactionCommand-Wert wird gedrückt

Für alle drei Fälle können Sie einen Action-Handler definieren, der aufgerufen wird, wenn das Ereignis eintritt.

Syntax UI-Code: **OnOpen** = **<Handler>**
OnClose = **<Handler>**
OnCommand = **<Handler>**
Schreiben: **<obj>.OnOpen** = **<Handler>**
<obj>.OnClose = **<Handler>**
<obj>.OnCommand = **<Handler>**

OnOpen

Der OnOpen-Handler wird gerufen, wenn die Dialogbox geöffnet wird. OnOpen-Handler müssen als **DialogAction** definiert sein, wobei der Parameter command unbestimmt ist und nicht verwendet werden sollte. Zum Zeitpunkt, an dem der R-BASIC-Handler ausgeführt wird, ist der Dialog bereits auf dem Schirm.

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

Beispiel UI-Code:

```
Dialog CommandDialog
  Caption$ = "Persönliche Daten"
  Children = NameText, VornameText, OKButton
  justifyChildren = J_CENTER_ON_CAPTION
  OnOpen = PslDataOpen
  END Object
```

Dazugehöriger BASIC-Code

```
DialogAction   PslDataOpen
  <.. Was immer hier zu tun ist ...>
  END Action
```

OnClose

Der OnClose-Handler wird gerufen, wenn die Dialogbox geschlossen wird. OnClose-Handler müssen als **DialogAction** definiert sein, wobei der Parameter command immer 1 (IC_CLOSE) ist. Zum Zeitpunkt, an dem der R-BASIC-Handler ausgeführt wird, ist der Dialog bereits nicht mehr auf dem Schirm.

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

Beispiel UI-Code:

```
Dialog CommandDialog
  Caption$ = "Persönliche Daten"
  Children = NameText, VornameText, OKButton
  justifyChildren = J_CENTER_ON_CAPTION
  OnClose = PslDataClose
  END Object
```

Dazugehöriger BASIC-Code

```
DialogAction   PslDataClose
  <.. Was immer hier zu tun ist ...>
  END Action
```

OnCommand

Der OnCommand-Handler wird gerufen, wenn ein Button mit einem **interaction-Command**-Wert, der nicht IC_CLOSE ist, angeklickt wurde (IC_CLOSE ruft den **OnClose**-Handler). OnCommand-Handler müssen als **DialogAction** definiert sein, wobei der Parameter "command" den **interactionCommand**-Wert des auslösenden Buttons enthält.

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

Ausführliches Beispiel:
Eine Dialog-Box mit 3 Schaltern. UI-Code:

```
Dialog ComplexDialog
  Caption$ = "Namen eingeben"
  Children = NameText, VornameText, MyReplyBar
  justifyChildren = J_CENTER_ON_CAPTION
  OnCommand = CommandHandler
  END Object

InputLine NameText
  Caption$ = "Name:"
  end object

InputLine VornameText
  Caption$ = "Vorname:"
  end object

Group MyReplybar
  MakeReplyBar
  Children = CloseButton, OKButton, DeleteButton
  END Object

Button CloseButton
  Caption$ = "Schließen"
  interactionCommand = IC_CLOSE
  END Object

Button DeleteButton
  Caption$ = "Löschen"
  interactionCommand = 1001    ' eigener Wert
  END Object

Button OKButton
  Caption$ = "Übernehmen"
  interactionCommand = IC_OK
  END Object
```

Im BASIC-Code muss nur der OnCommand-Handler vereinbart werden. Der CloseButton wird vom Dialog automatisch bedient, da er als interactionCommand IC_CLOSE gesetzt hat.

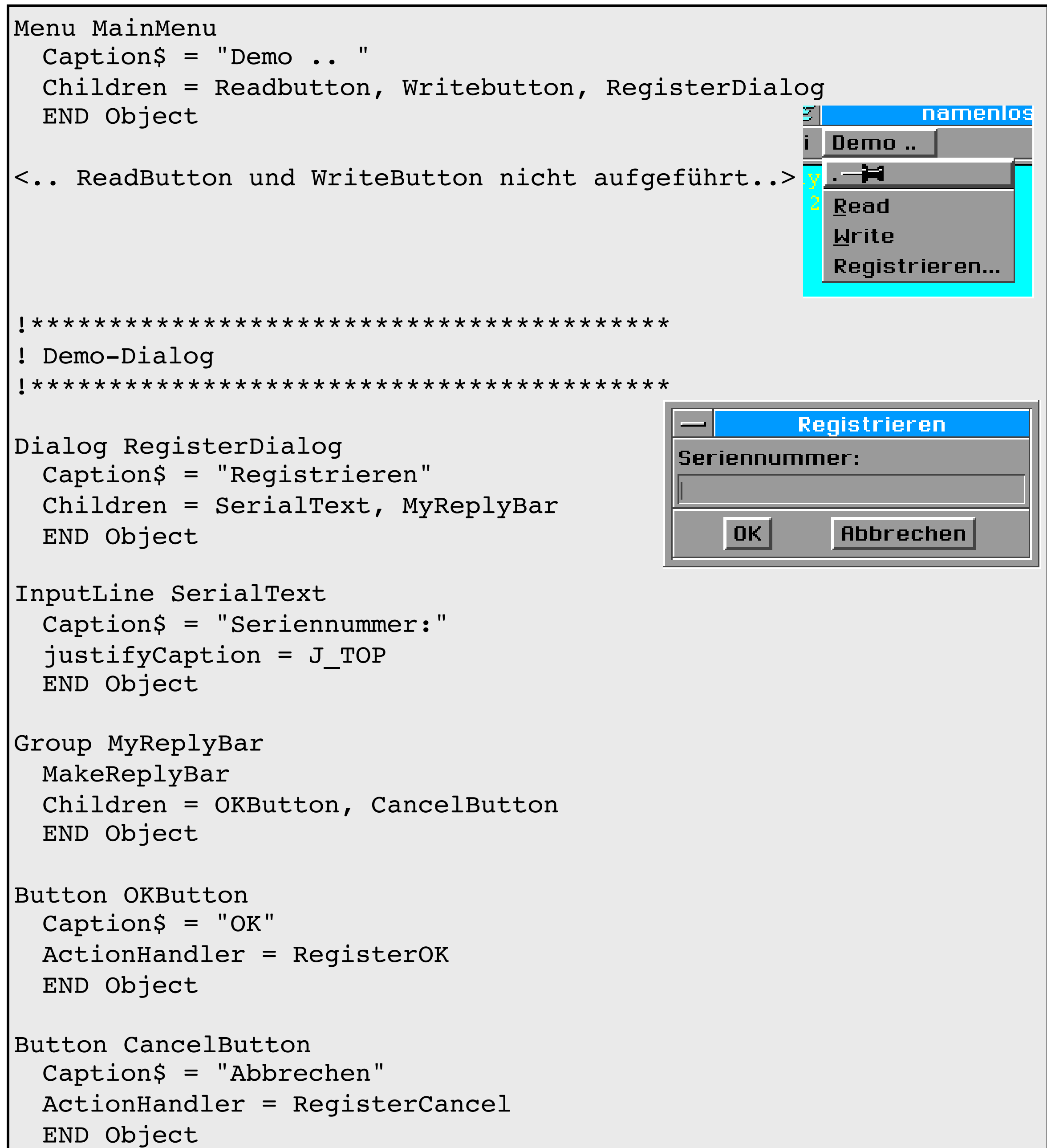
```
DialogAction CommandHandler
    IF command = 1001    THEN
     ' Texte löschen
     NameText.text$ = ""
     VornameText.text$ = ""
  END IF
  IF command = IC_OK   THEN
     MsgBox "Werte werden übernommen"
     ComplexDialog.Close
  END IF
END Action
```

4.6.5 Frei definierte Dialoge

dialogType = DT_NORMAL	keine systemerzeugten Buttons

Der wohl am einfachsten nachvollziehbare Weg, eine Dialogbox aufzubauen, ist, alle UI-Objekte selbst zu definieren, wie das folgende Beispiel zeigt:

UI-Code:




```
Menu MainMenu
  Caption$ = "Demo .. "
  Children = Readbutton, Writebutton, RegisterDialog
  END Object

<.. ReadButton und WriteButton nicht aufgeführt..>


!****************************************
! Demo-Dialog
!****************************************

Dialog RegisterDialog
  Caption$ = "Registrieren"
  Children = SerialText, MyReplyBar
  END Object

InputLine SerialText
  Caption$ = "Seriennummer:"
  justifyCaption = J_TOP
  END Object

Group MyReplyBar
  MakeReplyBar
  Children = OKButton, CancelButton
  END Object

Button OKButton
  Caption$ = "OK"
  ActionHandler = RegisterOK
  END Object

Button CancelButton
  Caption$ = "Abbrechen"
  ActionHandler = RegisterCancel
  END Object
```

Im Dialog-Objekt ist kein Wert für dialogType gesetzt, er steht per default auf DT_NORMAL.
Öffnet der Nutzer den Dialog kann er eine Seriennummer eingeben und dann auf OK oder Abbrechen klicken. Dadurch werden die Action-Handler RegisterOK bzw. RegisterCancel aufgerufen. Diese könnten so aussehen:

```
ButtonAction RegisterOK
  RegisterDialog.Close
  MsgBox "Registrierung erfolgreich"
  END Action

ButtonAction RegisterCancel
  RegisterDialog.Close
  END Action
```

Eine Behandlung der eingegeben Seriennummer wurde der Übersichtlichkeit halber ausgelassen. Wichtig ist, dass Sie die Dialogbox in beiden Fällen manuell schließen müssen. Das ist schon alles.

Der Dialog im Beispiel besteht aus 5 Objekten: dem Dialog-Objekt, einem Text-Objekt, einer ReplyBar und zwei Buttons. Reply-Bars sind typisch für Dialoge, deswegen kann das System automatisch eine Reply-Bar anlegen, wenn wir sie anfordern. Dazu muss man nur:

- Die Buttons als direktes Child des Dialogs festlegen
- Die Buttons mit der Anweisung **placeObject = REPLY_BAR** versehen.

Der folgende UI-Code erzeugt genau die gleiche Dialogbox wie der im Beispiel oben:

```
Dialog RegisterDialog
  Caption$ = "Registrieren"
  Children = SerialText, OKButton, CancelButton
  END Object

InputLine SerialText
  Caption$ = "Seriennummer:"
  justifyCaption = J_TOP
  END Object

Button OKButton
  placeObject = REPLY_BAR
  Caption$ = "OK"
  ActionHandler = RegisterOK
  END Object

Button CancelButton
  placeObject = REPLY_BAR
  Caption$ = "Abbrechen"
  ActionHandler = RegisterCancel
  END Object
```

4.6.6 Standard-Dialoge

Variable	Syntax im UI-Code	Im BASIC-Code
dialogType	dialogType = **numWert**	lesen, schreiben

Auch wenn es möglich ist, die UI-Objekte im Dialog vollständig selbst zu definieren, ist dies in vielen Fällen gar nicht nötig. Man kann stattdessen sogenannte "Standard-Dialoge" verwenden, die bereits einige vorgefertigte Objekte enthalten. Dies sind ein oder mehrere Buttons - Standard-Buttons genannt - und eine Reply-Bar. Da die Buttons direkt vom System erzeugt werden, können wir ihnen keinen Action-Handler zuweisen. Stattdessen belegt GEOS die Instance-Variable **interactionCommand** des Buttons.

Um einen Standard-Dialog zu verwenden benötigt man nur eine einzige Zeile im UI-Code: Die Belegung der Instance-Variablen **dialogType**.

Syntax UI-Code: **dialogType** = **numWert**
Lesen: **<numVar>** = **<obj>.dialogType**
Schreiben: **<obj>.dialogType** = **numWert**

Welche Buttons mit welchem **interactionCommand**-Wert erzeugt werden, hängt nur von der Belegung dieser Instance-Variablen ab. Die folgende Tabelle enthält die möglichen Werte für die Instance-Variable **dialogType** sowie die erzeugten Buttons und die zugeordneten interactionCommand-Werte.

dialogType	Wert	erzeugte Buttons
DT_NORMAL	0	keine (default, frei definierter Dialog)
DT_PROGRESS	2	"Stopp" oder "Anhalten" (IC_STOP)
DT_COMMAND	3	"Schließen" oder "Abbrechen" (IC_CLOSE)
DT_NOTIFICATION	4	"OK" (IC_OK)
DT_QUESTION	5	"Ja" (IC_YES) und "Nein" (IC_NO)
DT_DELAYED_APPLY	1	"Anwenden" bzw. "OK" (IC_APPLY) "Schließen" oder "Abbrechen" (IC_CLOSE)

Die genaue Beschriftung der Buttons kann je nach System-Version geringfügig variieren. Wenn nicht explizit anders angegeben wird die Dialogbox automatisch geschlossen, wenn der Nutzer auf einen der vom System erzeugten Buttons klickt.

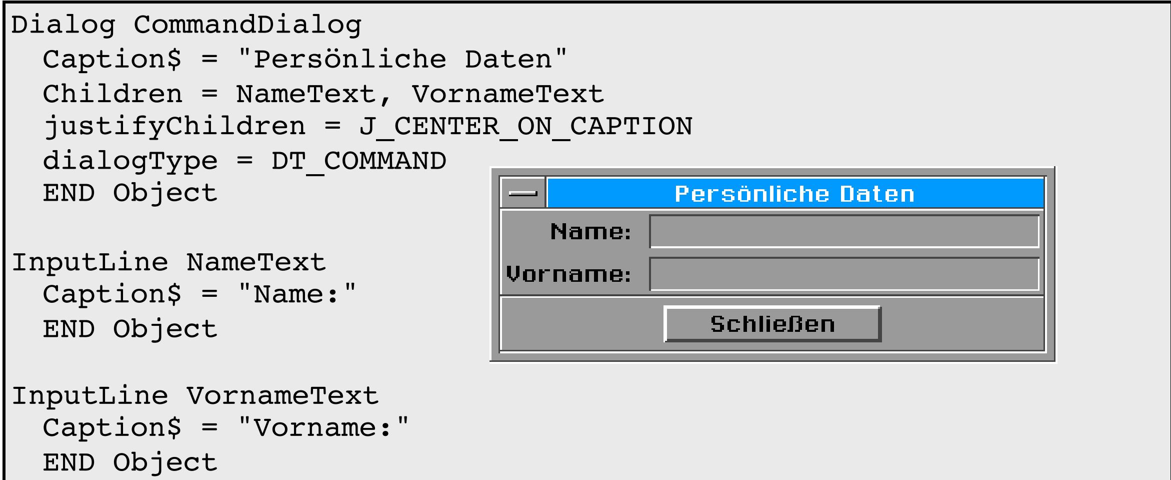
Die Verwendung der einzelnen dialogTypen und welche konkreten zusätzlichen Eigenschaften der Dialog dadurch erhält, wird in den nächsten Kapiteln ausführlich beschrieben.

4.6.6.1 Command-Dialoge

dialogType = DT_COMMAND	"Schließen" oder "Abbrechen" (IC_CLOSE)

Ein Command-Dialog (engl.: command = Kommando, Anweisung) erzeugt automatisch eine Reply-Bar mit einen Schließen-Button.

Beispiel UI-Code

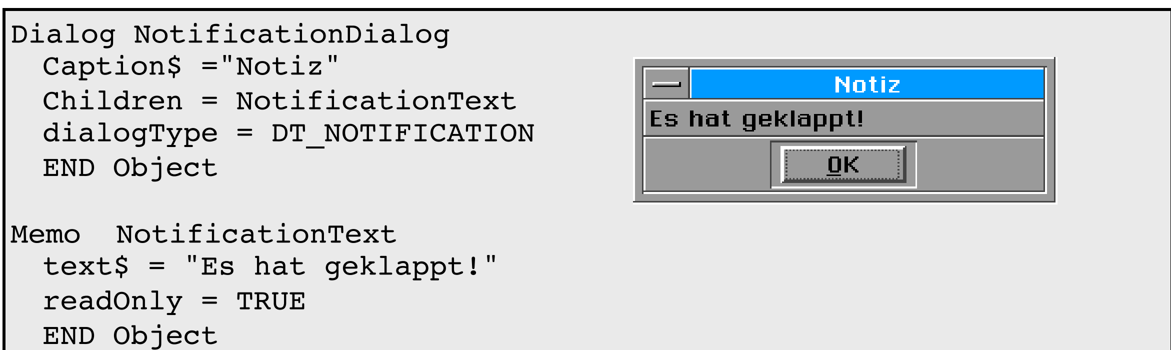



```
Dialog CommandDialog
  Caption$ = "Persönliche Daten"
  Children = NameText, VornameText
  justifyChildren = J_CENTER_ON_CAPTION
  dialogType = DT_COMMAND
  END Object

InputLine NameText
  Caption$ = "Name:"
  END Object

InputLine VornameText
  Caption$ = "Vorname:"
  END Object
```

4.6.6.2 Notification-Dialoge

dialogType = DT_NOTIFICATION	"OK" (IC_OK)

Notification-Dialoge (Hinweis-Dialoge) erzeugen automatisch eine Reply-Bar mit einen OK-Button. Sie werden sehr häufig für "Blocking-Dialoge" (attrs = DA_BLOCKING, Kapitel 4.6.7) oder für den "Information über.."-Dialog im Dateimenü verwendet. Bevor Sie einen Notification-Dialog programmieren sollten sie prüfen, ob einer der R-BASIC-Befehle **MsgBox, ErrorBox** oder **WarningBox** nicht bereits Ihren Anforderungen genügt. Sie sind intern als Blocking-Dialog mit dialogType = DT_NOTIFICATION realisiert.

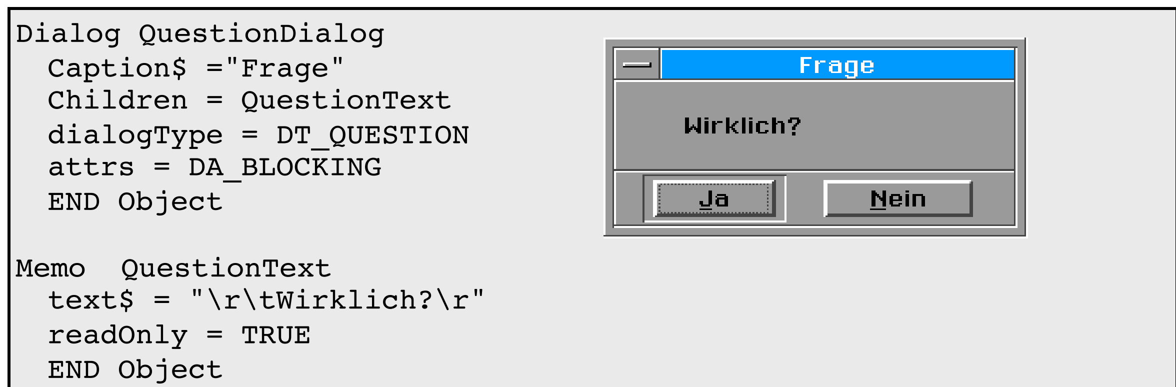



```
Dialog NotificationDialog
  Caption$ ="Notiz"
  Children = NotificationText
  dialogType = DT_NOTIFICATION
  END Object

Memo  NotificationText
  text$ = "Es hat geklappt!"
  readOnly = TRUE
  END Object
```

4.6.6.3 Question-Dialoge

dialogType = DT_QUESTION	"Ja" (IC_YES) und "Nein" (IC_NO)

Question-Dialoge (Frage-Dialoge) erzeugen automatisch eine Reply-Bar mit einen "Ja" und einem "Nein"-Button. Sie werden sehr häufig für "Blocking-Dialoge" (attrs = DA_BLOCKING, Kapitel 4.6.7) verwendet. Bevor Sie einen Question-Dialog programmieren sollten sie prüfen, ob der R-BASIC-Befehl **QuestionBox** nicht bereits Ihren Anforderungen genügt. QuestionBox ist intern als Blocking-Dialog mit dialogType = DT_QUESTION realisiert.



```
Dialog QuestionDialog
  Caption$ ="Frage"
  Children = QuestionText
  dialogType = DT_QUESTION
  attrs = DA_BLOCKING
  END Object

Memo  QuestionText
  text$ = "\r\tWirklich?\r"
  readOnly = TRUE
  END Object
```

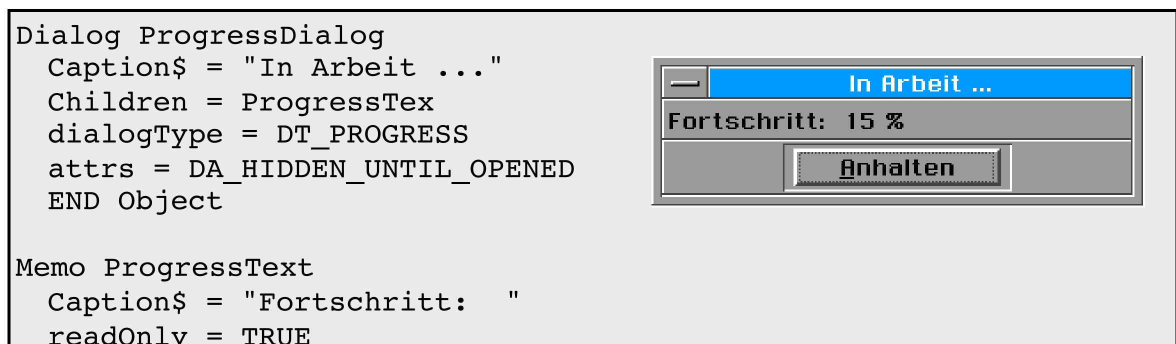

Tipp: Um herauszubekommen, ob der Nutzer auf "Ja" oder "Nein" geklickt hat können Sie den Dialog entweder als Blocking-Dialog programmieren (siehe Beispielcode und Kapitel 4.6.7) oder Sie verwenden den OnCommand-Handler des Dialogs (siehe Kapitel 4.6.4.3).

4.6.6.4 Progress-Dialoge

dialogType = DT_PROGRESS	"Stopp" oder "Anhalten" (IC_STOP)
interactionCommand	nur Lesen

Progress-Dialoge dienen dazu, dem Nutzer den Fortschritt einer Operation anzuzeigen und ein Abbrechen der Operation zu ermöglichen. Sie erzeugen automatisch eine Reply-Bar mit einen "Anhalten"-Button.

Im Beispiel wird zur Fortschrittsanzeige ein Textobjekt verwendet.



```
Dialog ProgressDialog
  Caption$ = "In Arbeit ..."
  Children = ProgressTex
  dialogType = DT_PROGRESS
  attrs = DA_HIDDEN_UNTIL_OPENED
  END Object

Memo ProgressText
  Caption$ = "Fortschritt:  "
  readOnly = TRUE
```

```
  fixedSize = 20 + ST_AVG_CHAR_WIDTH, 1 + ST_LINES_OF_TEXT
  END Object
```

Die korrekte Verwendung eines Progress-Dialogs erfordert etwas Aufmerksamkeit und Hintergrundwissen. GEOS ist ein Multithread-System, d.h. mehrere Prozesse (Threads) laufen quasi gleichzeitig ab. Selbst in einem R-BASIC-Programm gibt es zwei Threads - einen für den von Ihnen geschriebenen BASIC-Code und den UI-Thread, der die UI-Objekte bedient. Wenn Sie eine langwierige Operation ausführen wollen, z.B. das Suchen nach einer Datei, schreiben Sie dazu eine R-BASIC-Routine. Während diese läuft kann sie nicht durch das Anklicken eines Buttons unterbrochen werden, da der Action-Handler dieses Buttons auch im BASIC-Code-Thread läuft. Das Ereignis (Anklicken des Schalters "Abbrechen") würde vom System in eine Warteschlage gestellt und abgearbeitet, wenn die Suchroutine fertig ist. Sie können auf diese Weise also eine laufende Operation nicht unterbrechen.

An dieser Stelle kommt der UI-Thread und die Dialog-Instance-Variable **interactionCommand** ins Spiel. Klickt der Nutzer auf den "Anhalten" Button, so sendet dieser im UI-Thread - also parallel zur laufenden Suchroutine - eine Message an den Dialog. Der Dialog schließt sich uns setzt seine Instance-Variable **interactionCommand** auf den vom Button gesendeten Wert - in diesem Fall IC_STOP. Wenn Sie während der laufenden Operation regelmäßig die **interactionCommand**-Variable des Progress-Dialogs abfragen, können Sie den Prozess auf Anforderung abbrechen. Das folgende Code-Fragment zeigt, wie das geht:

```
ProgressDialog.Open   ' Dialog anzeigen

FOR N = 1  TO  100
  ProgressText.Text$ = Str$(n) + "  %"     ' Fortschritt melden

  < ... Nächsten Schritt der Operation durchführen .. >

  IF  ProgressDialog.interactionCommand = IC_STOP THEN BREAK
  NEXT N

ProgressDialog.Close ' Dialog schließen
IF  ProgressDialog.interactionCommand = IC_STOP    THEN
    MsgBox "Abgebrochen"
    END  IF
```

Tipp: Wenn es im konkreten Fall störend ist, dass sich der Dialog sofort selbständig schließt können Sie statt eines Progress-Dialogs einen frei definierten Dialog mit einem eigenen CancelButton verwenden. Dieser setzt zwar die interactionCommand-Variable des Dialogs, schließt ihn aber nicht, da er kein Standard-Button ist. Das ist im folgenden Beispiel gezeigt.

```
Dialog ProgressDialog
  Caption$ = "In Arbeit ..."
  Children = ProgressText, CancelButton
  attrs = DA_HIDDEN_UNTIL_OPENED
  END Object

Memo ProgressText
  Caption$ = "Fortschritt:  "
  readOnly = TRUE
  fixedSize = 20 + ST_AVG_CHAR_WIDTH, 1 + ST_LINES_OF_TEXT
  END Object

Button CancelButton
  Caption$ = "Anhalten"
  placeObject = REPLY_BAR
  interactionCommand = IC_STOP
  END Object
```


Sie dürfen nur nicht vergessen, den Dialog manuell mit ProgressDialog.Close zu schließen.

Anstelle eine Progress-Dialogs können Sie eventuell auch einen Button verwenden, dessen Instance-Variable **unhandledEvents** Sie abfragen. Das ist im Abschnitt 4.3 beschrieben.

4.6.6.5 Dialoge im Delayed Mode

dialogType = DT_DELAYED_APPLY	"Anwenden" bzw. "OK" (IC_APPLY) "Schließen" oder "Abbrechen" (IC_CLOSE)

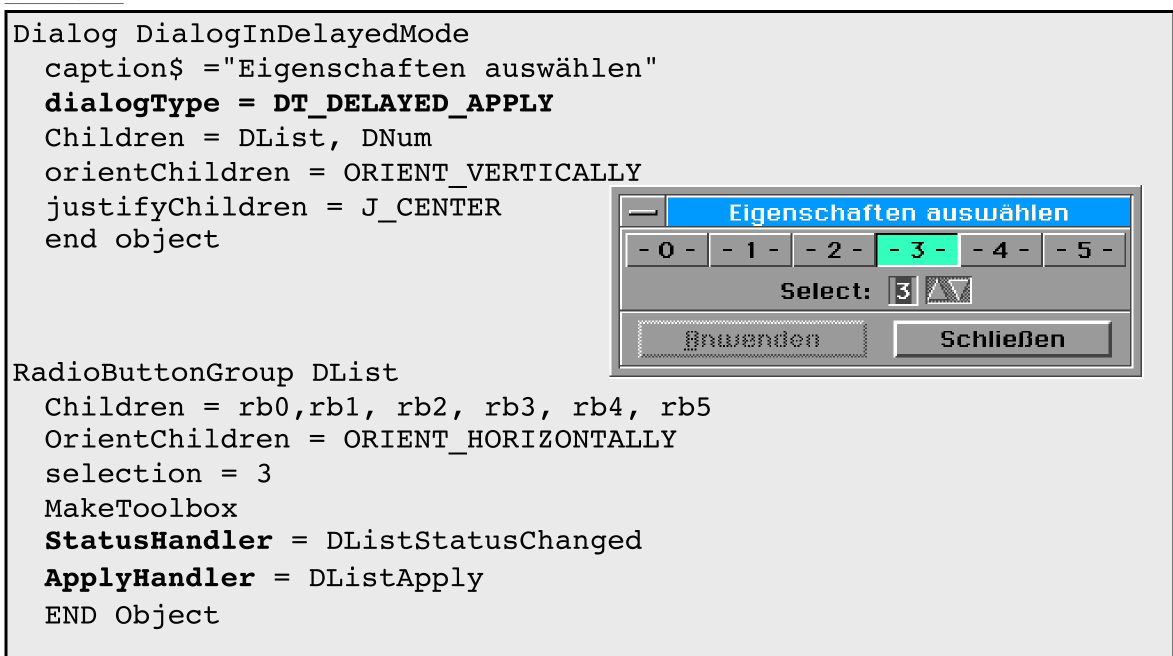
Der Dialogtyp DT_DELAYED_APPLY erzeugt eine Dialog-Box die im sogenannten "Delayed Mode" arbeitet, ganz so als würden Sie den Hint MakeDelayedApply für den Dialog setzen. Der Delayed Mode ist ausführlich im Kapitel 3.4.2 beschrieben. Im Kern besteht er in Folgendem:

- Die im Dialog enthaltenen Objekte (Texte, Number, Listen) senden statt ihrer Apply-Message (Aufruf des ApplyHandlers) zunächst eine Status-Message aus. Diese kann zur Kommunikation der Dialog-internen Objekte untereinander genutzt werden.
- Die Apply-Message wird erst ausgesendet, wenn der Nutzer auf den vom Dialog bereitgestellten Button "Anwenden" klickt. Dieses löst die Apply-Methode des Dialogs aus, die an alle seine Children weitergereicht wird (siehe Kapitel 3.4.1). Ein DT_DELAYED_APPLY-Dialog sorgt außerdem automatisch dafür, das der "Anwenden"-Button erst enabled wird, wenn mindestens eins der im Dialog enthaltenen Objekte geändert wurde.
- Die betroffenen Objekte (Texte, Number, Listen) müssen "modified" sein, damit sie ihre Apply-Message aussenden. Ändert der Nutzer das Objekt passiert das automatisch, ändern Sie das Objekt vom BASIC-Code aus, müssen Sie es selbst auf "modified" setzen.

Beispiel: Eine Dialogbox enthält eine Liste und ein Number-Objekt, die sich gegenseitig über Status-Handler auf dem neuesten Stand halten. Beim Klick auf "Anwenden" wird der ApplyHandler der Liste aufgerufen.

UI-Code

```
Dialog DialogInDelayedMode
  caption$ ="Eigenschaften auswählen"
  dialogType = DT_DELAYED_APPLY
  Children = DList, DNum
  orientChildren = ORIENT_VERTICALLY
  justifyChildren = J_CENTER
  end object

RadioButtonGroup DList
  Children = rb0,rb1, rb2, rb3, rb4, rb5
  OrientChildren = ORIENT_HORIZONTALLY
  selection = 3
  MakeToolbox
  StatusHandler = DListStatusChanged
  ApplyHandler = DListApply
  END Object
```



```
RadioButton rb0: Caption$ = " - 0 - ": identifier = 0: END Object
RadioButton rb1: Caption$ = " - 1 - ": identifier = 1: END Object
RadioButton rb2: Caption$ = " - 2 - ": identifier = 2: END Object
RadioButton rb3: Caption$ = " - 3 - ": identifier = 3: END Object
RadioButton rb4: Caption$ = " - 4 - ": identifier = 4: END Object
RadioButton rb5: Caption$ = " - 5 - ": identifier = 5: END Object

Number DNum
  Caption$ = "Select:"
  StatusHandler = DNumStatusChanged
  minVal = 0 : maxVal = 5
  value = 3
  END Object
```

BASIC-Code Die Zeile "`DList.modified = TRUE`" sorgt dafür, dass die Liste als "geändert" markiert wird, da sie sonst ggf. ihren Apply-Handler nicht aufruft.

```
LISTACTION DListApply
  MsgBox Str$(selection)+ " ist selektiert"
  END Action

LISTACTION DListStatusChanged
  DNum.value = selection
  END Action

NUMBERACTION DNumStatusChanged
  DList.selection = value
  DList.modified = TRUE
  END Action
```

Eine mögliche Ergänzung wäre, einen zusätzlichen Button "Zurücksetzen" hinzuzufügen, der die Liste und das Number-Objekt auf den Anfangsbestand zurücksetzt. Wie das gemacht wird, wird am Ende des nächsten Abschnitts (4.6.6.6) beschrieben.

4.6.6.6 Eigene Buttons in Standard-Dialogen

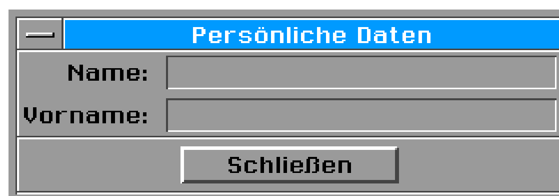
Sie können sowohl eigene Buttons zur Reply-Bar der Standard-Dialoge hinzufügen als auch die vorhandenen durch eigene ersetzen. Letzteres kann z.B. sinnvoll sein, wenn Sie die Beschriftung der Buttons ändern oder durch eine Grafik ersetzen wollen. In jedem Fall muss ein solcher Button die Zeile

```
  placeObject = REPLY_BAR
```

und einen **interactionCommand**-Wert enthalten. Existiert schon ein Standard-Button mit diesem **interactionCommand**-Wert, so wird er ersetzt, andernfalls wird ein weiterer Button hinzugefügt. Für ihren eigenen Button können Sie einen der vorhandenen **interactionCommand**-Werte verwenden (z.B. IC_OK) oder einen eigenen definieren. Eigene **interactionCommand**-Werte müssen größer als 999 sein, die Werte 0 bis 999 sind vom System reserviert!

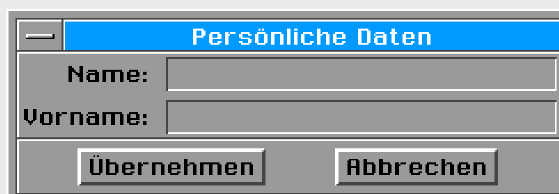
Beispiel: Der Command-Dialog aus dem letzten Abschnitt war so definiert (die Text-Objekte sind nicht mit aufgeführt):

```
Dialog CommandDialog
  Caption$ = "Persönliche Daten"
  Children = NameText, VornameText
  dialogType = DT_COMMAND
  justifyChildren = J_CENTER_ON_CAPTION
  END Object
```



Der Button mit der Aufschrift "Schließen" ist vom System erzeugt und entsprechend der Tabelle vorn (Anschnitt 4.6.6) mit dem interactionCommand IC_CLOSE belegt.
Um die Beschriftung des Buttons von "Schließen" auf "Abbrechen" zu ändern und einen weiteren Button mit der Aufschrift "Übernehmen" hinzuzufügen muss man folgendes tun (die Text-Objekte sind wieder nicht mit aufgeführt):

```
Dialog CommandDialog
  Caption$ = "Persönliche Daten"
  Children = NameText, VornameText, CloseButton, OKButton
  justifyChildren = J_CENTER_ON_CAPTION
  dialogType = DT_COMMAND
  OnCommand = DialogOKHandler
  END Object
```



```
Button CloseButton
  Caption$ = "Abbrechen"
  interactionCommand = IC_CLOSE         ' Button wird ersetzt
  placeObject = REPLY_BAR
  END Object

Button OKButton
  Caption$ = "Übernehmen"
  interactionCommand = IC_OK            ' Button wird hinzugefügt
  placeObject = REPLY_BAR
  END Object
```

Die beiden neuen Buttons bekommen einen interactionCommand-Wert, aber keinen ActionHandler. Um auf den OK-Button regieren zu können, bekommt der Dialog einen Action-Handler (OnCommand = DialogOKHandler). Dieser muss mindestens den Dialog schließen, da nicht-Standard-Buttons dies nicht automatisch tun.

Im BASIC-Code muss nur der OnCommand-Handler vereinbart werden:

```
DIALOGACTION DialogOKHandler
  CommandDialog.Close
  < .. Auswertung des Namens hier ..>
  END Action
```

Ein weiterer häufiger Fall für einen eigenen Button in einem Dialog-Objekt ist ein "Reset"-Button. Dafür kann man das InteractionCommand IC_RESET verwenden. Der Dialog muss dann einen OnCommand-Handler haben, der dieses Kommando auswertet und alle betroffenen Objekte auf ihren Anfangswert setzt.

Beispiel:

```
DIALOGACTION MyOnCommandHandler
  IF  command = IC_RESET  THEN
    NameText.text$ = ""
    VornameText.text$ = ""
    END IF
  < ... >
  END Action
```

4.6.7 Arbeit mit Blocking-Dialogen

attrs = DA_BLOCKING	Auf Reaktion warten
numVar = **OpenBlockingDialog**(<dialogObj>)	Dialog aktivieren

Oftmals ist es für den Programmablauf erforderlich, dass der Nutzer zuerst den Dialog bedient, bevor die Programmabarbeitung fortgesetzt werden kann. Das heißt, die weitere Programmabarbeitung wird solange blockiert ("Blocking"), bis der Dialog beendet ist. Ein Beispiel wäre die Nachfrage, ob die Daten gespeichert werden sollen oder nicht.

Für Blocking-Dialoge kann man sowohl frei definierte als auch Standard-Dialoge verwenden.

attrs = DA_BLOCKING

Ein Blocking-Dialog muss die Zeile

```
attrs = DA_BLOCKING
```

im UI-Code gesetzt haben (ein späteres setzen im BASIC-Code ist möglich, aber meist nicht sinnvoll). Diese Zeile bewirkt folgendes:

- Das System erzeugt keinen "Aktivierungsbutton" für den Dialog, d.h. er kann nicht über ein Menü geöffnet werden.
- Der Dialog **muss** mit der Funktion **OpenBlockingDialog()** (siehe unten) geöffneten werden. Die Methoden Open und OpenNoDisturb führen zu einem Laufzeitfehler.
- Der Dialog ist immer modal (siehe Kapitel 4.6.2). Wenn Sie keinen Wert für die Instance-Variable "modal" vorgeben, wird APP_MODAL genommen, NON_MODAL wird ignoriert (R-BASIC setzt dann APP_MODAL).

Wichtig! Vergessen Sie nicht, den Dialog an geeigneter (irgendeiner) Stelle in den generic Tree einzubinden.

OpenBlockingDialog()

GEOS ist eine Multi-Thread-System. Selbst in einem R-BASIC Programm laufen zwei Threads (Prozesse) gleichzeitig: Der Code-Thread, der den von Ihnen geschrieben BASIC-Code ausführt und der UI-Thread, der die UI-Objekte bedient.
Die Funktion **OpenBlockingDialog()** öffnet einen Blocking-Dialog. Der BASIC-Code Thread wird blockiert ("schlafen" gelegt, er verbraucht auch keine CPU-Zeit mehr), es läuft nur noch der UI-Thread. Dadurch kann der Nutzer die Objekte im Dialog bedienen (Listenelemente auswählen, Texte eingeben etc.). Um den Dialog zu beenden muss der Nutzer auf einen Schalter klicken, der einen **interactionCommand**-Wert gesetzt hat (vgl. Kapitel 4.6.4.2). Daraufhin kehrt OpenBlockingDialog() zurück und liefert den **interactionCommand**-Wert des Buttons, der betätigt wurde. Der BASIC-Code Thread wird fortgesetzt und kann den Wert auswerten.
Achtung! OpenBlockingDialog liefert den Wert Null, wenn GEOS heruntergefahren wird, während ein Blocking-Dialog offen ist.

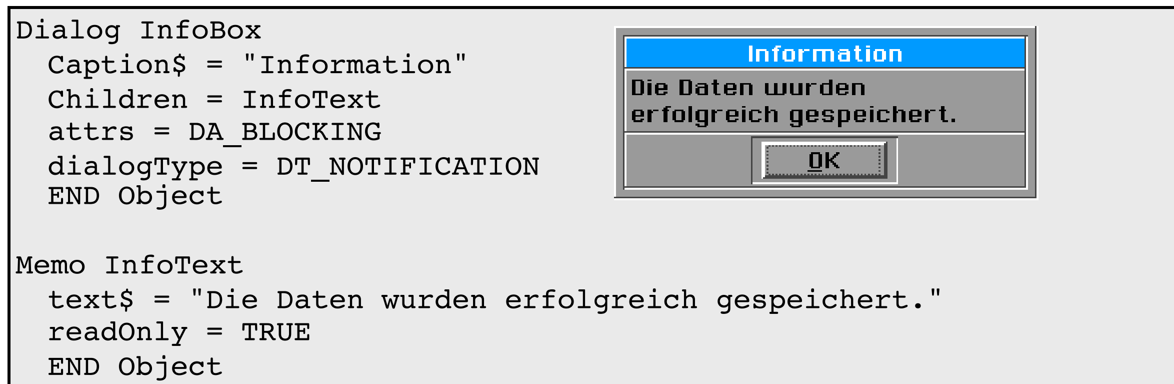
Schließen von Blocking-Dialogen

In den meisten Fällen, insbesondere wenn Sie einen der Standard-Dialog-Typen gewählt haben (z.B. **dialogType** = DT_QUESTION, siehe Kapitel 4.6.6), schließt sich der Dialog automatisch. Sollte das nicht der Fall sein (z.B. weil Sie einen selbst definierten interactionCommand-Wert verwendet haben), müssen Sie den Dialog selbst schließen.

```
  cmd = OpenBlockingDialog( SaveFilesDialog )
  SaveFilesDialog.Close
  IF  cmd = 1001 THEN ....
```

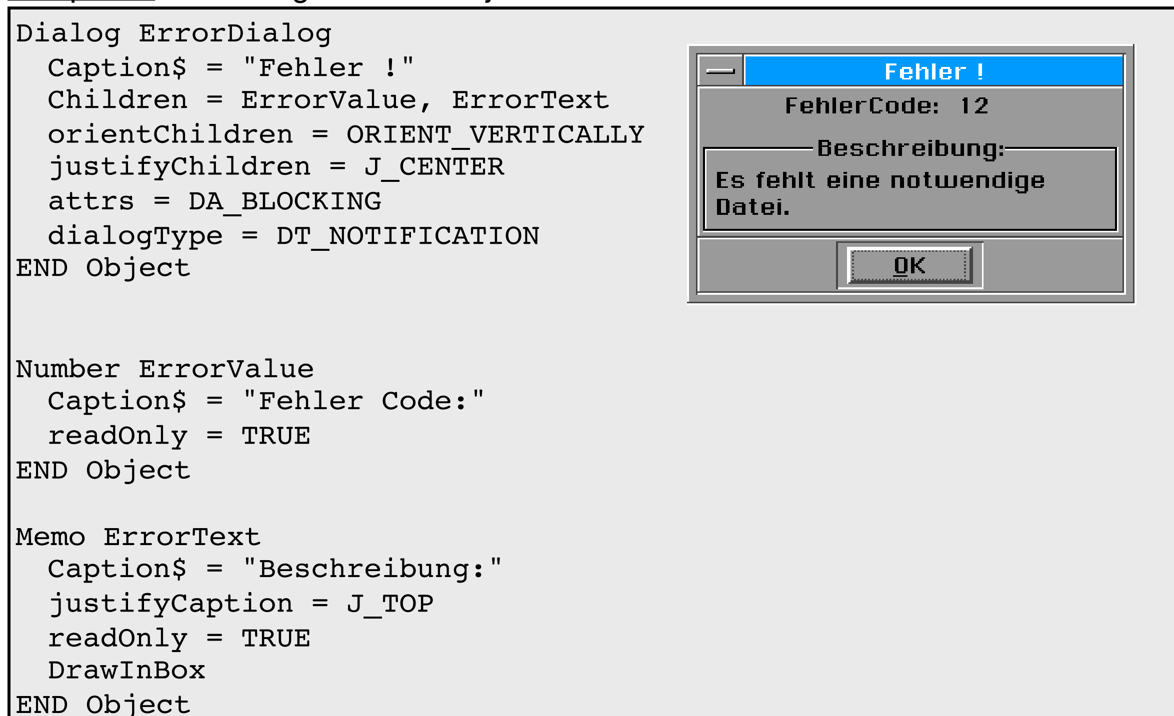
Sie müssen, wie im Beispiel, den zurückgegebenen Wert "cmd" nicht überprüfen, da es erlaubt ist, die Close-Methode auch aufzurufen, wenn es gar nicht nötig wäre.

Beispiel 1: Ein einfacher Dialog. Er dient zur Verdeutlichung des Prinzips. An seiner Stelle könnte man auch den BASIC-Befehl **MsgBox** verwenden.



```
Dialog InfoBox
  Caption$ = "Information"
  Children = InfoText
  attrs = DA_BLOCKING
  dialogType = DT_NOTIFICATION
  END Object

Memo InfoText
  text$ = "Die Daten wurden erfolgreich gespeichert."
  readOnly = TRUE
  END Object
```


Beispiel 2: Ein Dialog mit zwei Objekten.



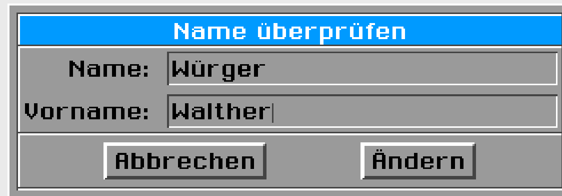
```
Dialog ErrorDialog
  Caption$ = "Fehler !"
  Children = ErrorValue, ErrorText
  orientChildren = ORIENT_VERTICALLY
  justifyChildren = J_CENTER
  attrs = DA_BLOCKING
  dialogType = DT_NOTIFICATION
END Object


Number ErrorValue
  Caption$ = "Fehler Code:"
  readOnly = TRUE
END Object

Memo ErrorText
  Caption$ = "Beschreibung:"
  justifyCaption = J_TOP
  readOnly = TRUE
  DrawInBox
END Object
```

Der Dialog wird in einer SUB verwendet:

```
SUB ShowError(err as integer)
DIM retVal
  ErrorValue.value = err
  ErrorText.text$ = "Es fehlt eine notwendige Datei."
  retVal = OpenBlockingDialog( ErrorDialog )
END SUB
```

Beispiel 3: Eine Dialog-Box mit 2 frei definierten Schaltern. Die Reply-Bar wird automatisch erzeugt, da die Buttons die Zeile "placeObject = REPLY_BAR" enthalten. UI-Code:

```
Dialog NameDialog
  Caption$ = "Name überprüfen"
  Children = NameText, VornameText, CloseButton, OKButton
  justifyChildren = J_CENTER_ON_CAPTION
  attrs = DA_BLOCKING
  END Object

InputLine NameText
  Caption$ = "Name:"
  end object

InputLine VornameText
  Caption$ = "Vorname:"
  end object

Button CloseButton
  Caption$ = "Abbrechen"
  placeObject = REPLY_BAR
  interactionCommand = 1001
  END Object

Button OKButton
  Caption$ = "Ändern"
  placeObject = REPLY_BAR
  interactionCommand = 1002
  END Object
```



Im BASIC-Code der SUB "CheckName" wird der Dialog initialisiert, aufgerufen und dann ausgewertet:

```
' globale Variablen
DIM  gName$, gVorname$

<.. irgendwo im Code ..>
gName$ = "Würger"
gVorname$ = "Wilhelm"


SUB CheckName()
DIM cmd   AS Word
  NameText.text$ = gName$
```

```
  VornameText.text$ = gVorname$

  cmd = OpenBlockingDialog ( NameDialog )
  NameDialog.Close
  IF cmd = 1002 THEN
    ' Werte auslesen
    gName$ = NameText.text$
    gVorname$ = VornameText.text$
    END IF

END SUB
```

Ergänzende Hinweise

- Objekte in Blocking-Dialogen dürfen keinerlei Action-Handler oder Status-Handler haben. Das gilt für das Dialog-Objekt selbst, die Buttons, Listen-Objekte usw. Insbesondere können DynamicList-Objekte nicht in Blocking-Dialogen verwendet werden, da sie einen Query-Handler benötigen. Der Grund dafür ist einfach: Blocking-Dialoge blockieren den BASIC-Code Thread. Die Action-Handler werden aber in diesem Thread ausgeführt - sie können also nicht behandelt werden, solange der Dialog offen ist.

- Blocking-Dialoge müssen mindestens einen Button mit einem **interaction-Command**-Wert haben. Dies ist der einzige Weg einen Blocking-Dialog zu verlassen. Vergessen Sie das, hängt GEOS.

- Häufig werden für Blocking-Dialoge Standard-Dialog-Typen (dialogType = DT_xxx) benutzt. Sie können aber auch beliebige eigene Buttons / interactionCommand-Werte definieren (vgl. Kapitel 4.6.6.6)

- Sollte eine Dialogbox mit dem Aufruf von OpenBlockingDialog() nicht erscheinen, haben Sie wahrscheinlich vergessen, das Dialog-Objekt in den generic Tree einzubinden. Das ist ein sehr häufiger Fehler.

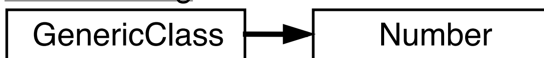
- Wird GEOS heruntergefahren, während ein Blocking-Dialog offen ist, so kehrt OpenBlockingDialog() zurück, der Dialog schließt sich und als Interaction-Command wird der Wert Null geliefert. Der gerade laufende BASIC-Code (sehr oft irgendein Action-Handler) wird zu Ende geführt und danach das R-BASIC-Programm geschlossen. Erst nachdem alle Programme geschlossen sind fährt GEOS endgültig herunter. Was hat das mit Blocking-Dialogen zu tun? Ganz einfach: Sie müssen immer damit rechnen, dass OpenBlockingDialog() statt denen von Ihnen vorgegebene interactionCommand-Werte den Wert Null liefert. Es sollte in diesem Fall weder eine potentiell gefährliche Aktion ausgelöst werden noch eine Endlos-Schleife erzeugt werden.

4.7 Number

Objekte der Klasse Number (engl.: Zahl) dienen dazu, Zahlen darzustellen oder einzugeben. Obwohl sie sehr einfach zu verwenden sind, verfügen Sie über einen hohen Grad an "Eigenintelligenz".

Die folgenden Ausführungen gehen zunächst grundsätzlich davon aus, dass das Number-Objekt im normalen Modus (nicht im sogenannten "Delayed Mode") arbeitet. Das ist der Normalfall, wenn man nicht spezielle Hints setzt, um in den Delayed Mode zu kommen. Dieser "Delayed Mode" ist ausführlich im Kapitel 3.4.2 (Delayed Mode und Status-Message) dieses Handbuchs beschrieben.

Abstammung



Spezielle Instance-Variablen

Variable	Syntax im UI-Code	Im BASIC-Code
value	value = **numWert**	lesen, schreiben
minVal	minVal = **numWert**	lesen, schreiben
maxVal	maxVal = **numWert**	lesen, schreiben
incVal	incVal = **numWert**	lesen, schreiben
ApplyHandler	ApplyHandler = **<Handler>**	nur schreiben
StatusHandler	StatusHandler = **<Handler>**	nur schreiben
modified	modified = TRUE I FALSE	lesen, schreiben
NavigateToNextFieldOnReturn	NavigateToNextFieldOnReturn	—
displayFormat	displayFormat = **numWert**	lesen, schreiben
decimal	decimal = **numWert**	lesen, schreiben
look	look = **numWert**	lesen, schreiben
sliderShowIntervals	sliderShowIntervals = main [, sub]	lesen, schreiben
SliderNoDigitalDisplay	SliderNoDigitalDisplay	—
SliderShowMinMax	SliderShowMinMax	—

Methoden:

Methode	Aufgabe
Increment	value-Wert um incVal erhöhen
Decrement	value-Wert um incVal verringern
Apply	Apply-Handler aufrufen
SendStatus	Status-Handler aufrufen

Action-Handler-Typen:

Handler-Typ	Parameter
NumberAction	(sender as object, value as real)

4.7.1 Grundlegende Eigenschaften

R-BASIC Number-Objekte speichern eine Zahl im Wertebereich von –32768 ... +32767 mit einer Genauigkeit von 4 Stellen nach dem Komma. Das gilt für die Instance-Variablen **value**, **minVal**, **maxVal** und **incVal**. Um Kommastellen darstellen zu können muss die Instance-Variable **displayFormat** auf einen passenden Wert gesetzt werden (siehe Kapitel 4.7.2), per Default werden ganze Zahlen dargestellt.

Der Zahlenwert

value	value = **numWert**	lesen, schreiben
minVal	minVal = **numWert**	lesen, schreiben
maxVal	maxVal = **numWert**	lesen, schreiben
incVal	incVal = **numWert**	lesen, schreiben

value

Die Instance-Variable **value** enthält die vom Number-Objekt dargestellte Zahl. Sie sollten den Wert auf die notwendige Anzahl von Stellen runden (R-BASIC Befehl ROUND), wenn Sie ihn vom Number-Objekt lesen, da ein Number-Objekt eine begrenzte Genauigkeit in der internen Zahlendarstellung aufweist.

Syntax UI- Code: **value** = **numWert**
Lesen: **<numVar>** = **<obj>** . **value**
Schreiben: **<obj>.value** = **numWert**

minVal

Die Instance-Variable **minVal** enthält den unteren Grenzwert für die Instance-Variable value. Das Number-Objekt sorgt selbständig dafür, dass dieser Grenzwert nicht unterschritten wird. Der Defaultwert für **minVal** beträgt 0.

Syntax UI- Code: **minVal** = **numWert**
Lesen: **<numVar>** = **<obj>** . **minVal**
Schreiben: **<obj>.minVal** = **numWert**

maxVal

Die Instance-Variable **maxVal** enthält den oberen Grenzwert für die Instance-Variable value. Das Number-Objekt sorgt selbständig dafür, dass dieser Grenzwert nicht überschritten wird. Der Defaultwert für **maxVal** beträgt 32766.

Syntax	UI- Code:	**maxVal** = numWert
	Lesen:	<numVar> = <obj> . **maxVal**
	Schreiben:	<obj>.**maxVal** = numWert

incVal

Die Instance-Variable **incVal** enthält den Wert, um den die Instance-Variable **value** erhöht bzw. erniedrigt wird, wenn man auf einen der Pfeile neben dem Zahlenwert klickt. Der Defaultwert für **incVal** beträgt 1.

Syntax	UI- Code:	**incVal** = numWert
	Lesen:	<numVar> = <obj> . **incVal**
	Schreiben:	<obj>.**incVal** = numWert

Anwenden der Änderungen

ApplyHandler	ApplyHandler = <Handler>	nur schreiben
modified	modified = TRUE I FALSE	lesen, schreiben
NavigateToNextFieldOnReturn	NavigateToNextFieldOnReturn	—

ApplyHandler

Die Instance-Variable **ApplyHandler** enthält den Namen des ActionHandlers, der aufgerufen wird, wenn das Number-Objekt seinen Wert "anwenden" will (engl. to apply: Anwenden). Dieser muss als **NumberAction** vereinbart sein. Der Apply-Handler wird üblicherweise aufgerufen, wenn der Nutzer auf die Pfeile des Number-Objekts klickt, nach Eingabe eines Wertes auf "Enter" drückt oder einen Slider "zieht". Sliders werden im Kapitel 4.7.3 besprochen.
Der Wert für den **ApplyHandler** wird üblicherweise im UI-Code gesetzt. Bei Bedarf kann er auch zur Laufzeit (im BASIC-Code) gesetzt, aber nicht gelesen werden.

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

Beispiel: Ein typisches Number-Objekt

```
Number testNumber
  Caption$ = "Value = "
  ApplyHandler = numberApply
  incVal = 5
  value = 280
  minVal = -105
  maxVal = 300
  END Object
```

BASIC-Code des Apply-Handlers

```
NumberAction numberApply
  Print ROUND(value)
  END Action
```

Beachten Sie die Verwendung der R-BASIC Funktion ROUND( ), um sicherzustellen, dass der ausgegeben Wert ganzzahlig ist. Auch wenn das Number-Objekt eine Ganze Zahl darstellt kann es trotzdem intern eine Zahl mit Nachkommastellen gespeichert haben.

Hinweis: Es ist möglich den ApplyHandler des Number-Objekts manuell (vom BASIC-Code aus) zu aktivieren. Dazu wird die von der GenericClass geerbte Methode **Apply** verwendet. Da ApplyHandler nur ausgelöst werden, wenn das Objekt "modified" ist, muss es vorher als "modified" markiert werden. Alternativ könnte man dem Objekt auch den Hint *ApplyEvenIfNotModifie*d geben.

Beispiel:

```
  testNumber.modified = TRUE
  testNumber.Apply
```

Eine ausführliche Beschreibung dazu finden Sie im Kapitel 3.4 (Die "Apply"-Message) dieses Handbuchs.

modified

Die Instance-Variable modified enthält die Information, ob der vom Number-Objekt dargestellte Wert seit dem letzten Aufruf des Apply-Handlers geändert wurde (modified = TRUE) oder nicht (modified = FALSE).

Beachten Sie, dass ein Verändern des Objekts vom BASIC-Code aus (z.B. Belegen der Instance-Variable **value**), das Objekt nicht als "modified" markiert, d.h. der Wert der Instance-Variablen **modified** wird nicht verändert. Sie können dies bei Bedarf selbst machen, indem Sie die Anweisung "<obj>.modified = TRUE" verwenden.

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

Wenn Sie die Instance Variable **modified** lesen, werden Sie feststellen, dass sie Null enthält, es sei denn, Sie haben sie explizit auf einen anderen Wert gesetzt. Ändert der Nutzer nämlich den Zahlenwert, so passiert intern folgendes:

- Die Instance-Variable **modified** wird mit TRUE belegt.
- Es wird geprüft ob ein ApplyHandler vorhanden ist und dieser wird ggf. aufgerufen.
- Die Instance-Variable **modified** wird zurückgesetzt (mit FALSE belegt).

Hinweis: Im sogenannten Delayed Mode (siehe entsprechendes Kapitel weiter unten) werden die letzten beiden Schritte nicht ausgeführt, so dass die Instance-Variable **modified** eine eigene Bedeutung erhält.

NavigateToNextFieldOnReturn

Drückt der Nutzer in Eingabefeld eines Number-Objekts die Enter-Taste bleibt der Cursor üblicherweise dort stehen. Der Hint **NavigateToNextFieldOnReturn** bewirkt, dass der Cursor stattdessen zum nächsten Eingabefeld weiterrückt.

Syntax UI-Code: **NavigateToNextFieldOnReturn**

Methoden

Increment	value-Wert um incVal erhöhen
Decrement	value-Wert um incVal verringern

Increment

Die Methode **Increment** bewirkt, dass der Instance-Wert value um incVal erhöht wird, genau so, als ob der Nutzer auf den entsprechenden Pfeil des Number-Objekts geklickt hat.

Syntax BASIC-Code: <obj>.**Increment**

Decrement

Die Methode **Decrement** bewirkt, dass der Instance-Wert value um incVal vermindert wird, genau so, als ob der Nutzer auf den entsprechenden Pfeil des Number-Objekts geklickt hat.

Syntax BASIC-Code:	<obj>.**Decrement**

4.7.2 Display-Format

displayFormat	displayFormat = **numWert**	lesen, schreiben
decimal	decimal = **numWert**	lesen, schreiben

Grundsätzlich gilt: Number-Objekte speichern Zahlen im Wertebereich von –32768 ... +32767 mit einer Genauigkeit von 4 Stellen nach dem Komma. Diese Zahl kann aber von einem Number-Objekt auf verschieden Weise dargestellt werden. Das wird über die Instance-Variablen **displayFormat** und **decimal** gesteuert.

decimal

Die Instance-Variable **decimal** stellt die Anzahl der angezeigten Nachkommastellen ein. Voraussetzung ist, dass das mit **displayFormat** eingestellte Anzeigeformat auch Nachkommastellen zulässt. Das ist bei allen Formaten außer DF_INTEGER der Fall. Der Default-Wert für **decimal** ist 3, der Maximalwert ist 4.

Syntax	UI- Code:	**decimal** = **numWert**
	Lesen:	**<numVar>** = **<obj>** . **decimal**
	Schreiben:	**<obj>.decimal** = **numWert**

displayFormat

Die Instance-Variable **displayFormat** stellt ein, auf welche Weise die vom Number-Objekt gespeicherte Zahl dargestellt wird. Der Default-Wert für **displayFormat** ist DF_INTEGER.

Syntax	UI- Code:	**displayFormat** = **numWert**
	Lesen:	**<numVar>** = **<obj>** . **displayFormat**
	Schreiben:	**<obj>.displayFormat** = **numWert**

Das Number-Objekt rechnet dabei den internen Wert in die das gewünschte Format um. Intern geht das Number-Objekt davon aus, das der intern gespeicherte Wert in US-Points vorliegt. Es stehen die folgenden Formate zur Verfügung:

Konstante	Wert	Darstellung (Beispiele)
DF_INTEGER	0	Ganzzahlig, z.B. 12
DF_DECIMAL	1	Mit Dezimalstellen, z.B. 28,6
DF_POINTS	2	US-Point 720 Pt
DF_INCHES	3	Inches 18,9 in
DF_CENTIMETERS	4	Zentimeter 12,3 cm
DF_MILLIMETERS	5	Millimeter 534 mm
DF_PICAS	6	Picas 60 pt
DF_EUR_POINTS	7	Europäische Points 98 ep
DF_CICEROS	8	Ciceros 78,9 ci
DF_POINTS_OR_MILLIMETERS	9	72 Pt oder 25,4 mm
DF_INCHES_OR_CENTIMETERS	10	1,5 in oder 3,81 cm

Anmerkungen zu einigen der Konstanten

DF_INTEGER

Der Wert wird ganzzahlig ohne Einheit dargestellt. Ist intern ein nicht ganzzahliger Wert gespeichert, so wird der dargestellte Wert gerundet.

DF_DECIMAL

Der Wert wird zur Darstellung auf den durch die Instance-Variable **decimal** vorgegebene Genauigkeit gerundet.

DF_POINTS_OR_MILLIMETERS
DF_INCHES_OR_CENTIMETERS

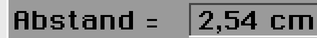
Auf Systemen, deren lokalen Einstellungen auf "US-Einheiten" basieren, wird der Wert in Points oder Inches dargestellt, sind die Einheiten auf "Metrisch" gestellt, erfolgt die Darstellung in mm bzw. cm.

Beachten Sie, dass mit **displayFormat** wirklich nur das Anzeige-Format geändert und der Wert zur Anzeige umgerechnet wird. Das bedeutet konkret, dass der in der Instance-Variablen **value** gespeicherte Wert bei den meisten Formaten nicht mit dem Zahlenwert in der Anzeige identisch ist. Wie oben schon erwähnt geht das Number-Objekt davon aus, das der intern gespeicherte Wert in US-Points vorliegt.
Dabei gelten die folgenden Umrechnungen:

Konstante	Definition	Umrechnung zu US-Point (Pt)	
DF_INTEGER		keine Umrechnung nötig	
DF_DECIMAL		keine Umrechnung nötig	
DF_POINTS		keine Umrechnung nötig	
DF_INCHES		1 in = 72 Pt	1 Pt = 1/72 in
DF_CENTIMETERS	1 in = 2.54 cm	1 cm ≈ 28.246 Pt	1 Pt ≈ 0.03278 cm
DF_MILLIMETERS	10 mm = 1 cm	1 mm ≈ 2.8246 Pt	1 Pt ≈ 0.3278 mm
DF_PICAS	1 pt = 1/6 in	1 pt = 12 Pt	1 Pt = 1/12 pt
DF_EUR_POINTS		1 ep ≈ 1.0656 Pt	1 Pt = 0.93844 ep
DF_CICEROS	1 ci = 12 ep	1 ci ≈ 12,7872 Pt	1 Pt ≈ 0,078203 ci

Beispiel: value steht auf 72 (Pt), das sind 2,54 cm

```
Number testnumber
  Caption$ = "Abstand = "
  displayFormat = DF_CENTIMETERS
  value = 72
  incVal = 36
  maxVal = 3000
  END Object
```



Beachten Sie, dass beim Lesen der Instance-Variable **value** natürlich der dort gespeicherte Wert gelesen wird. Im Beispiel ist das 72, und nicht etwa 2,54!

4.7.3 Angepasstes Aussehen und Sliders

Variable	Syntax im UI-Code	Im BASIC-Code
look	look = **numWert**	lesen, schreiben
sliderShowIntervals	sliderShowIntervals = main [, sub]	lesen, schreiben
SliderNoDigitalDisplay	SliderNoDigitalDisplay	—
SliderShowMinMax	SliderShowMinMax	—

look

Das Aussehen eine Number-Objekts kann mit der Instance-Variable look angepasst werden (engl. look : Aussehen, äußere Erscheinung).

Syntax UI- Code: **look** = **numWert**
Lesen: **<numVar>** = **<obj>** . **look**
Schreiben: **<obj>.look** = **numWert**

Dabei stehen die folgenden Werte zur Verfügung:

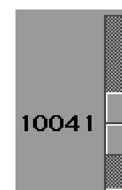
Konstante	Wert	Aussehen
LOOK_NORMAL	0	komplett "normal"
LOOK_NOT_DIGITALLY_EDITABLE	1	Eingabefeld read-only
LOOK_NOT_INCREMENTABLE	2	keine Pfeile
LOOK_X_SLIDER	3	Schieberegler, horizontal
LOOK_Y_SLIDER	4	Schieberegler, vertikal

LOOK_NORMAL

LOOK_NOT_DIGITALLY_EDITABLE

LOOK_NOT_INCREMENTABLE

LOOK_X_SLIDER



LOOK_Y_SLIDER

Für Sliders gibt es weitere Möglichkeiten, das Aussehen anzupassen. Sie sollten die folgenden Hints nur anwenden, wenn die Instance-Variable look auf einen der SLIDER-Werte gesetzt ist.

sliderShowIntervals

Dieser Hint versieht den Slider mit Intervall-Marken. Sie können angeben, wie viele Hauptintervalle gezeichnet und ob diese noch unterteilt werden sollen.

Syntax	UI- Code:	**sliderShowIntervals** = **main** [, **sub**]
	Lesen:	**<numVar>** = **<obj>** . **sliderShowIntervals** (**n**) n = 0: main-Wert lesen n = 1: sub-Wert lesen
	Schreiben:	**<obj>.sliderShowIntervals** = **main** [, **sub**] **<obj>.sliderShowIntervals** = 0 löscht den Hint aus den Instance-Daten, d.h. die Intervalle werden entfernt.

Der Parameter **main** gibt die Anzahl der Hauptintervalle an. Der Parameter **sub** ist optional und legt fest, in wie viele Unterintervalle die Hauptintervalle zu unterteilen sind. Die Sub-Intervallstriche sind etwas kürzer als die der Hauptintervalle. Die Einteilung in Intervalle erfolgt völlig unabhängig vom dargestellten Zahlenbereich.
Beispiel: siehe unten

SliderNoDigitalDisplay

Dieser Hint entfernt den Zahlenwert, der üblicherweise über bzw. neben dem Slider erscheint.

Syntax	UI- Code:	**SliderNoDigitalDisplay**

Beispiel: siehe unten

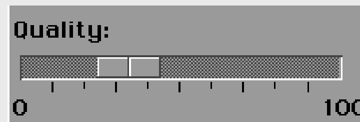
SliderShowMinMax

Dieser Hint fügt die Anzeige des Minimal- und Maximal-Wertes (**minVal** und **maxVal**) zum Slider hinzu. Die Werte werden in dem durch die Instance-Variable **displayFormat** spezifizierten Format angezeigt.

Syntax	UI- Code:	**SliderShowMinMax**

Beispiel: Ein Slider mit 4 Intervallen, die in je 2 Unterintervalle geteilt sind. Die Anzeige des aktuellen Digitalwertes ist deaktiviert, die Anzeige des Minimum- und Maximum-Wertes ist aktiviert.

```
Number QualitySlider
  Caption$ = "Quality:"
  justifyCaption = J_TOP
  ApplyHandler = numberApply
  value = 30
  incVal = 10
  minVal = 0
  maxVal = 100
  look = LOOK_X_SLIDER
  sliderShowIntervals = 4, 2
  SliderNoDigitalDisplay
  SliderShowMinMax
  END Object
```



Beachten Sie, dass Number-Objekte von der GenericClass abstammen und daher alle Eigenschaften und Fähigkeiten dieser Klasse erben. Insbesondere gilt das für die Geometrie-Hints sowie die Möglichkeit sie auf "**enabled** = FALSE" oder "**readOnly** = TRUE" zu setzen. Die Bilder zeigen "**readOnly** = TRUE" Objekte, links einen Slider mit dem Hint **SliderNoDigitalDisplay** gesetzt und rechts ein "normales" Objekt.

4.7.4 Weitere Hinweise zur Arbeit mit Number-Objekten

- Die Instance-Variablen **value**, **minVal**, **maxVal** und **incVal** sind aus der Sicht von R-BASIC vom Datentyp REAL. Intern verwenden Number-Objekte zur Speicherung der Werte den Datentyp WWFixed. Er besteht aus einem Integer-Wert (Wertebereich von –32768 bis +32767) für den ganzzahligen Teil und einem WORD für den gebrochenen Teil.

- Daraus ergeben sich die folgenden Einschränkungen:
  - Die Werte für **value**, **minVal**, **maxVal** und **incVal** können in einem Bereich von –32768 bis +32767 liegen.
  - Die Genauigkeit ist auf 4 Stellen nach dem Komma begrenzt.
  - Es muss "minVal – incVal >= –32768" und "minVal + incVal <= 32767" gelten.

- Setzen Sie den Wert für **value** zur Laufzeit, prüft das Number-Objekt, ob er im Bereich von **minVal** bis **maxVal** liegt. Ist das nicht der Fall, so passt das Number-Objekt den Wert ohne Warnmeldung an. Fehlerhafte Werte für **minVal** bzw. **maxVal** werden auf den größtmöglichen bzw. kleinstmöglichen Wert gesetzt.

- Das Number-Objekt rundet den internen Wert auf die zur Anzeige notwendige Stellenzahl. Sie daher sollten den **value**-Wert auch auf die notwendige Anzahl von Stellen runden (R-BASIC Befehl ROUND), wenn Sie ihn vom Number-Objekt lesen oder einen Number-Action-Handler programmieren. Wegen der begrenzten Genauigkeit in der internen Zahlendarstellung weicht der intern dargestellte Wert oft geringfügig vom angezeigten Wert ab.

- Die Geometrie-Hints ExpandWidth, ExpandHeight, DivideWidthEqually und DivideHeightEqually werden nur in den ~_SLIDER - Looks unterstützt. Bei Bedarf packen Sie das Number-Objekt (mit den anderen Looks) in eine Group, die die entsprechenden Hints gesetzt hat.

Interne Details

Der von Number-Objekten zur Speicherung von Zahlen verwendete Datentyp WWFixed wird im GEOS System sehr häufig, unter anderem für alle grafischen Berechnungen, verwendet. Das liegt daran, dass sie einen sehr guten Kompromiss aus guter Genauigkeit (Nachkommastellen) und sehr hoher Rechengeschwindigkeit ermöglicht. Die oben genannte Begrenzung des Wertebereichs ergibt sich aus der Verwendung eine Integer-Wertes für den ganzzahligen Teil. Der Nachkommateil kann als Anzahl der 1/65536 interpretiert werden, die auf den ganzzahligen Teil zu addieren ist. Da 1/65536 ≈ 0,000015 ist, kann der Nachkommawert nur in dieser Abstufung geändert werden. Eine Genauigkeit von 5 Nachkommastellen ist damit nicht mehr erreichbar. Selbst 4 Stellen sind mit Vorsicht zu genießen, da 0,0001 intern bereits als 7/65536 dargestellt werden muss, was aber eigentlich 0,0001068 ist. R-BASIC begrenzt den Wert für **decimal** daher auf 4.

Die Bedingungen "minVal - incVal >= –32768" und "minVal + incVal <= 32767" haben folgenden Grund: Erhöht oder Erniedrigt das Number-Objekt seinen value-Wert um den durch incVal gegebenen Wert (klicken Sie z.B. auf den Hoch- bzw. Runter-Pfeil), so nutzt es dazu Ganzzahl-Arithmetik. Daher kann es zu einer Wertebereichsüberschreitung (Übertrag) kommen, die vom GEOS-System aus Performance-Gründen nicht behandelt wird. R-BASIC beachtet diese Bedingungen beim Compilieren des UI-Codes, aber - ebenfalls aus Performance-Gründen - nicht zur Laufzeit. Sie können daher gegen diese beiden Bedingungen verstoßen, wenn Sie die Instance-Variablen zur Laufzeit setzen. Sie werden schon sehen, was Sie davon haben.

Wenn Sie eine höhere Genauigkeit wünschen oder den Integer-Wertebereich verlassen müssen, können Sie zur Eingabe einer Zahl einer der Befehle **INPUT** oder **InputBox** oder ein Text-Objekt (z.B. Memo oder InputLine) und ggf. die BASIC-Funktionen **VAL**, **ValLocal** bzw. **Str$** verwenden.

4.7.5 Number-Objekte im Delayed Mode

Ein Number-Objekt kann im "Delayed Mode" (engl.: verzögerter Modus) arbeiten. Dazu muss man dem Objekt selbst bzw. einem seiner Parents im UI-Code den Hint **MakeDelayedApply** geben oder man bindet das Objekt als Child in einem Dialog ein, dessen **dialogType** Instance Variable auf DT_DELAYED_APPLY gesetzt ist. Dieser "Delayed Mode" ist ausführlich im Kapitel 3.4.2 (Delayed Mode und Status-Message) dieses Handbuchs beschrieben, eine Beschreibung des Dialog-Objekts im Delayed Mode finden Sie im Kapitel 4.6.6.5.

Instance Variable	Syntax im UI-Code	Im BASIC-Code
StatusHandler	StatusHandler = **<Handler>**	nur schreiben

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

Der **StatusHandler** wird im Delayed Mode statt des ApplyHandlers gerufen, wenn der Nutzer auf die Pfeile des Number-Objekt klickt, nach Eingabe eines Wertes auf "Enter" drückt oder einen Slider "zieht". Der ApplyHandler hingegen wird erst auf Anforderung gerufen (siehe Kapitel 3.4.2).

Die Instance-Variable **modified** kann TRUE enthalten, nämlich dann, wenn das Objekt vom User modifiziert wurde, der ApplyHandler aber noch nicht gerufen wurde. Der Aufruf des ApplyHandlers setzt auch im Delayed Mode den modified-Status zurück.

Methode	Aufgabe
SendStatus	Status-Handler aufrufen

Syntax BASIC-Code: **<obj>.SendStatus**

Die Methode **SendStatus** fordert das Objekt auf, seinen StatusHandler aufzurufen (d.h. seine Status-Message zu senden).

(Leerseite)

(Leerseite)