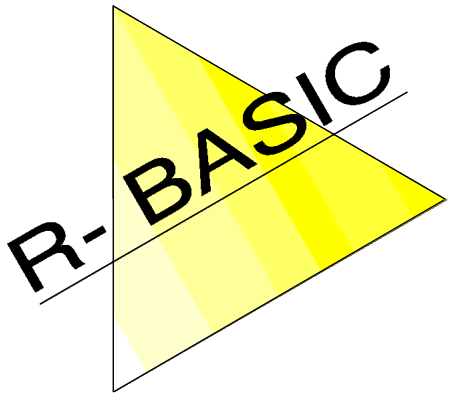
R-BASIC

Einfach unter PC/GEOS programmieren



Programmierhandbuch

Volume 5
Weitere Funktionen, Libraries

Version 1.0

(Leerseite)

Inhaltsverzeichnis

(Leerseite)

2.11 Weitere Funktionen

2.11.1 Versionsnummern

Da sowohl die R-BASIC IDE als auch ein R-BASIC Programm aus verschiedenen Komponenten bestehen gibt es verschiedene Arten von Versionsnummern.

- Das BASIC-Programm selbst hat eine Versionsnummer. Sie kann mir dem Befehl Version$ abgefragt werden.
- Jede GEOS-Datei hat eine Protocol- und eine Release-Nummer. Diese können abgefragt und auch geändert werden.
- Die Versionsnummer der R-BASIC IDE, mit der ein Programm erstellt wurde kann mit dem BASIC Befehl BasicVersion$ abgefragt werden.

Version$

Jedes Programm und jede Library besitzt eine Versionsnummer, mit deren Hilfe man verschiedene Programmversionen identifizieren kann. Man kann sie über die globale Variable Version$ ermitteln. Rufen Sie Version$ aus einer Library heraus auf so erhalten Sie die Versionsnummer der Library.

Syntax **<stringVar>** = **Version$**

Der Aufbau von Version$ ist folgender:

Major.Minor.Change

Die beiden ersten Werte (Major und Minor) werden vom Programmierer festgelegt und sind insbesondere bei Libraries von großer Bedeutung (siehe unten). Der dritte Wert (Change) wird bei jedem Compilerlauf automatisch hochgezählt. Änderungen des Major- oder des Minor-Wertes setzen den Change-Wert wieder auf Null.
Verwechseln Sie Version$ nicht mit der Variablen BasicVersion$, welche die Versionsnummer der BASIC-IDE enthält, die das aktuelle Programm compiliert hat.

Die Werte für Major und Minor können im Menüpunkt "Programm" geändert werden. Der Major-Wert sollte geändert (vergrößert) werden, wenn wesentliche neue Funktionen dazugekommen sind, ein neuer Wert für Minor sollte Bugfixes oder kleinere Änderungen anzeigen.

• Version eines Programms
  Sie sind in der Wahl der Versionsnummer völlig frei.

• Version einer Library
Wenn Sie eine BASIC-Library schreiben, die nur für genau ein spezielles BASIC-Programm gedacht ist, spielt die Versionsnummer keine große Rolle. Sie lassen die Versionsnummer bei 0.0, dann gibt es keine Probleme.

Wenn Sie eine BASIC-Library schreiben, die von verschiedenen Programmen verwendet werden soll, müssen Sie der Versionsnummer eine gewisse Aufmerksamkeit schenken. Beim Compilieren eines Programms, das ihre Library benutzt, wird die aktuelle Versionsnummer der Library mit abgespeichert, damit das Programm später entscheiden kann, ob es mit der aktuell vorhandenen Library-Version zusammenarbeiten kann oder nicht. Dazu werden der Major und der Minor-Wert herangezogen. Der Change-Wert wird ignoriert.
Ein Programm, das mit einer Library-Version 2.3 compiliert wurde, arbeitet mit späteren Versionen der Library (z.B. Version 2.5 oder 3.1) zusammen. Es wird aber weder mit der Version 2.1 noch mit der Version 1.9 der Library zusammenarbeiten.

Tipps:
- Während der Entwicklungsphase einer Library oder eines Programms sollten Sie die Versionsnummer nicht ändern, egal was Sie tun.
- Häufig ist es so, dass während der Entwicklung der Major-Wert auf Null bleibt. Bei der ersten Veröffentlichung der Library oder des Programms setzt man die Versionsnummer auf 1.0.
- Jedes Mal wenn eine neue Version einer Library veröffentlicht wird sollten Sie die Versionsnummer ändern.

Release- und Protokollnummer

Auf GEOS Systemebene werden die Protokoll- und Releasenummer verwendet, um zu prüfen, ob Dateien, Libraries und Programme zueinander kompatibel sind.

Eine komplette Beschreibung der Befehle zur Arbeit mit Protocol- und Releasenummern finden Sie im Handbuch "Spezielle Themen", Kapitel 9.2 (Dateiattribute). Die folgende Tabelle enthält einen Überblick.

Befehl / Struktur	Aufgabe / Bedeutung
FileGetRelease	GEOS Releasenummer auslesen
FileSetRelease	GEOS Releasenummer ändern
FileGetProtocol	GEOS Protokollnummer auslesen
FileSetProtocol	GEOS Protokollnummer ändern
ReleaseNumber	Struktur zum Speichern einer Protocol- oder Releasenummer

BasicVersion$

Sie können die Versionsnummer der BASIC-IDE, die das aktuelle Programm compiliert hat, mit der Systemvariablen BasicVersion$ abfragen. Nicht zu verwechseln mit der Variablen Version$, die die Versionsnummer des aktuell ausgeführten Programms enthält.

Syntax **<stringVar>** = **BasicVersion$**

BasicVersion$ liefert einen String der Form:

Major.Minor.Change-Engeneering

z.B. 1.0.1-17

Dieser Wert ist der gleiche, wie er im Menü "Datei-Information" angezeigt wird, es handelt sich um die Release-Nummer der R-BASIC IDE.

Interne Details

Der folgende Abschnitt enthält Hintergrundinformationen, die zur täglichen Arbeit mit R-BASIC nicht unbedingt benötigt werden.

Die Releasenummer enthält die "Version" eines Programms oder einer Library. Sie wird angezeigt, wenn Sie im GeoManager die Tastenkombination Strg-G eingeben bzw. den Menüpunkt "Datei" -> "Info & Attribute" anklicken.
Die Protokollnummer beschreibt die "Fähigkeiten" eines Programms oder einer Library bzw. die "innere Struktur" einer Datei. Sie wird zur Versionsprüfung auf Systemebene verwendet. Wenn die Protokollnummer nicht passt ist die Datei (Dokument, Library, Programm) nicht kompatibel. Zum Beispiel verwendet die R-BASIC IDE die Protokollnummer um zu entscheiden ob R-BASIC die Datei öffnen und bearbeiten kann.

R-BASIC verwendet die Releasenummer zur Versionsprüfung. Aus Sicht des GEOS-Systems sind sowohl die R-BASIC Codedatei als auch die BIN-Datei Dokumente. Das gilt auch für R-BASIC Libraries. Das GEOS-System ignoriert die Releasenummer von Dokumenten, deshalb kann R-BASIC sie benutzen.

Der R-BASIC Compiler setzt beim Compilieren die ersten drei Felder der Releasenummer der BIN-Datei entsprechend der Versionsnummer des Programms bzw. der Library. Das hat folgende Konsequenzen:

- Der Wert kann mit dem BASIC Befehl Version$ ausgelesen werden.
- Uni-Installer ab Version 1.2 verwendet die Releasenummer von R-BASIC Dateien um zu entscheiden ob eine Datei neuer ist als eine andere Datei mit gleichem Namen. Dadurch kann verhindert werden, dass ältere Versionen eines R-BASIC-Programms oder einer R-BASIC Library eine neuere Version überschreiben.

- Bei R-BASIC Libraries wird die Releasenummer (und damit die Versionsnummer der Library) verwendet um zu entscheiden, ob ein BASIC Programm mit dieser Library zusammenarbeiten kann.

Beim Anlegen eines eigenständigen Programms (R-App) wird die Releasenummer des Launchers auf die Versionsnummer des Programms gesetzt. Dadurch kann der Nutzer die Versionsnummer des Programms im GeoManager mit der Tastenkombination Strg-G erfahren.

2.11.2 Arbeit mit Datum und Zeit

R-BASIC unterstützt die Grundfunktionen zum Zugriff auf das Systemdatum sowie elementare Funktionen zur Anzeige von Datum und Uhrzeit. Außerdem wird die Arbeit mit dem Julianischen Datum unterstützt.

Alle Zeitfunktionen, einschließlich der Funktionen zum Zugriff auf das Dateidatum benutzen eine Struktur, die **DateAndTime** heißt.

```
STRUCT DateAndTime
   year, month, day        AS  INTEGER
   hour, minute, second    AS  INTEGER
   END  STRUCT
```

year, month, day enthalten das Jahr (z.B. 2014), den Monat (1...12) und den Tag (1 ... 31).
hour, minute, second enthalten die Stunde (0..23), die Minute (0...59) und die Sekunde (0...59).

Zugriff auf Datum und Zeit

SysGetTime

SysGetTime liefert das aktuelle Systemdatum und die Systemzeit.

Syntax: **<time>** = **SysGetTime**()
<time>: Variable vom Typ **DateAndTime**.

```
Beispiel:
DIM time  AS  DateAndTime
time = SysGetTime()
Print "Das Weltall. Unendliche Weiten. \
Wir schreiben das Jahr ";time.year
```

SysGetCount

SysGetCount liefert die Anzahl der "Tics" (1/60 s) seit dem letzten Systemstart.

Syntax: **<numVar>** = **SysGetCount**()
<numVar>: numerische Variable

Formatierung von Datum und Zeit

Da man über die DateAndTime-Struktur direkten Zugriff auf das Jahr, den Monat, die Stunde usw. als Zahl hat, ist es nicht kompliziert einen formatierten String zu erzeugen, der Datum oder Zeit enthält. Zum Beispiel erzeugt der Code

```
s$ = "Jahr: " + Str$(time.year)
```

einen String, der das Jahr als Text enthält.
Aber manchmal braucht man einen einfachen Weg Datums- oder Zeitangaben zu formatieren. Die folgenden Routinen erledigen das.

FormatDate$

FormatDate$ gibt das Datum in der Form "24.03.2010" oder in der Form "2010/03/24" aus. Die lokalen Einstellungen des Computers werden dabei nicht berücksichtigt.

Syntax: **<stringVar>** = **FormatDate$**(**time** [, **flag**])
time: Variable (oder Funktion) vom Typ DateAndTime.
flag (optional): TRUE: Das Format "2010/03/24" benutzen
FALSE (default): Das Format "24.03.2010" benutzen

Beispiel:

```
DIM time  AS  DateAndTime
time = SysGetTime()
Print  "Aktuelles Datum: "; FormatDate$(time)
```

FormatTime$

FormatTime$ gibt die Uhrzeit im 24-Stunden-Format aus. Die lokalen Einstellungen des Computers werden dabei nicht berücksichtigt.

Syntax: **<stringVar>** = **FormatTime$**(**time** [, **flag**])
time: Variable (oder Funktion) vom Typ **DateAndTime**.
flag (optional): TRUE (default): Sekunden anzeigen (17:34:54)
FALSE: Sekunden nicht anzeigen (17:34)

Beispiel:

```
DIM time  AS  DateAndTime
time = SysGetTime()
Print  "Aktuelle Uhrzeit: "; FormatTime$(time)
```

Weekday$, DayOfWeek

Weekday$ gibt den Wochentag in Textform aus, wobei die lokalen Einstellungen (Sprache) des Computers berücksichtigt werden. DayOfWeek liefert eine Zahl (0..6) die dem Wochentag entspricht. Der Sonntag hat den Wert Null.

Syntax: **<stringVar>** = **Weekday$**(**time**)
Syntax: **<numVar>** = **DayOfWeek**(**time**)
time: Variable (oder Funktion) vom Typ **DateAndTime**.

Beispiel:

```
DIM time  AS  DateAndTime
time = SysGetTime()
Print  "Heute ist "; Weekday$( time )
Print  "Das ist der"; DayOfWeek( time ) + 1 ; ". Tag der Woche."
```

LocalFormatDateAndTime$

LocalFormatDateAndTime$ greift auf die Systemeinstellung für die Darstellung von Datum und Zeit zu und formatiert Datums- und Zeitangaben entsprechend diesen Einstellungen. Dadurch erscheinen die Datums- und Zeitangaben in dem vom Nutzer eingestellten Format. Das kann allerdings auf verschiedenen Systemen völlig unterschiedlich sein.

Syntax: **<stringVar>** = **LocalFormatDateAndTime$**(**time**, **format**)
time: Variable (oder Funktion) vom Typ **DateAndTime**.
format: Eine DateAndTimeFormat (DTF-) Konstanten aus der aus der Tabelle unten.

Die folgende Tabelle zeigt das Ausgabeformat von LocalFormatDateAndTime$ für die Standardeinstellungen auf einem deutschen PC/GEOS System. Auf anderen Systemen, insbesondere auf fremdsprachigen Systemen, kann das Ausgabeformat völlig anders aussehen.

Konstante	Wert	Ausgabe
DTF_LONG	0	Sonntag, 28. Februar 2010
DTF_LONG_CONDENSED	1	So, 28. Feb. 2010
DTF_LONG_NO_WEEKDAY	2	28. Februar 2010
DTF_LONG_NO_WEEKDAY_CONDENSED	3	28. Feb. 2010
DTF_SHORT	4	28.02.10
DTF_ZERO_PADDED_SHORT	5	28.02.10
DTF_MD_LONG	6	Sonntag, 28. Februar
DTF_MD_LONG_NO_WEEKDAY	7	28. Februar
DTF_MD_SHORT	8	28.02.
DTF_MY_LONG	9	Februar 2010
DTF_MY_SHORT	10	02.10
DTF_YEAR	11	2010
DTF_MONTH	12	Februar
DTF_DAY	13	28.
DTF_WEEKDAY	14	Sonntag
DTF_HMS	15	23:48:52
DTF_HM	16	23:48
DTF_H	17	23
DTF_MS	18	48:52
DTF_HMS_24HOUR	19	23:48:52
DTF_HM_24HOUR	20	23:48

Julianisches Datum

Das Julianische Datum ist eine Zahl, die der Anzahl der Tage seit dem 1.1.4713 vor Christus, 12 Uhr mittags entspricht. Man beginnt 12 Uhr mittags damit bei astronomischen Beobachtungen während der Nacht kein Datumswechsel auftritt. Die Nachkommastellen entsprechen der Uhrzeit (bezogen auf 12 Uhr mittags).

Das Julianische Datum ist fortlaufend. Unregelmäßigkeiten wie unterschiedliche Monatslängen, Monats und Jahreswechsel oder auch Schaltjahre werden automatisch berücksichtigt. Damit lassen sich Fragen wie "Wie viele Tage liegen zwischen dem 28.04.2016 und dem 17.07.1962?" einfach durch die Differenz der beiden zugehörigen Julianischen Datumszahlen beantworten. Der Vergleich von zwei Zeitpunkten (einschließlich der Frage, welcher früher liegt) reduziert sich auf den Vergleich von zwei Real-Zahlen.
Für das Julianische Datum wird allgemein die Abkürzung JD verwendet.

JDFromDAT

Die Funktion JDFromDAT ermittelt den julianischen Datumswert (JD) aus einer DateAndTime Struktur (DAT).

Syntax: **<numVar>** = **JDFromDAT** (**<dat>**)
<numVar>: numerische Variable, empfohlener Datentyp: Real
<dat>: Variable oder Ausdruck vom Datentyp DateAndTime

DATFromJD

Die Funktion DATFromJD rechnet einen julianischen Datumswert (JD) in eine DateAndTime Struktur um.

Syntax: **<var>** = **DATFromJD** (**<jd>**)
<jd>: numerischer Ausdruck. Julianisches Datum
<var>: Variable vom Datentyp DateAndTime

Das folgende Beispiel gibt aus, welches Datum wir in 100 Tagen, gerechnet ab heute, haben.

```
DIM jetzt, dann AS DateAndTime
DIM jd as Real
  jetzt = SysGetTime()
  jd = JDFromDAT(jetzt)
  dann = DATFromJD ( jd + 100)

  Print "In 100 Tagen ist der ";FormatDate$(dann)
```

Wir wollen wissen in wie vielen Tagen Weihnachten ist.

```
DIM zeit as DateAndTime
DIM ist, soll, count as Real
  zeit = SysGetTime()
  ist = JDFromDAT(zeit)       ' aktuelles Julianisches Datum

  zeit.day = 24
  zeit.month = 12
  soll = JDFromDAT(zeit)      ' Julianisches Datum vom Weihnachten

  count = soll - ist
  Print "In";count;"Tagen ist Weihnachten."
```

JDDeltaFromMinutes

Die Funktion JDDeltaFromMinutes berechnet, wie groß die Differenz zweier Julianischer Daten ist, die sie sich um die gegebene Anzahl von Minuten

unterscheiden. Damit kann man Daten und Uhrzeiten ermitteln, die um eine gegebene Zeitdifferenz vor oder nach einem bekannten Zeitpunkt liegen.

Syntax: **<numVar>** = **JDDeltaFromMinutes** (**<n>**)
<numVar>: numerische Variable, empfohlender Datentyp: Real
<n>: Zeitdifferenz in Minuten. Negative Zahlen und Dezimalzahlen sind zulässig.
Beispiel: 1 min, 30 sek.: time = 1.5

Beispiele:
Wir wollen wissen wie spät es in 73,5 Minuten ist

```
DIM zeit as DateAndTime
DIM ist, soll, count as Real
  zeit = SysGetTime()
  ist = JDFromDAT(zeit)       ' aktuelles Julianisches Datum
  soll = ist + JDDeltaFromMinutes(73.5)
  zeit = DATFromJD(soll)      ' DateAndTime-Struktur bilden
  Print "In 73,5 min ist es ";FormatTime$(zeit);" Uhr"
```

Die folgende Function liefert TRUE, wenn sich zwei Daten um mehr als 1 Stunde unterscheiden. Als Vergleichswert wählen wir 60.001 Minuten (ca. 1/10 Sekunde mehr als 1 h) weil es durch die interne Zahlendarstellung zu minimalen Abweichungen kommen kann.

```
FUNCTION CompareTime(dat1, dat2 AS DateAndTime) as Real
DIM jd1, jd2, diff as Real
  jd1 = JDFromDAT(dat1)
  jd2 = JDFromDAT(dat2)
  if ( jd1 > jd2 ) THEN
    diff = jd1 - jd2
  ELSE
    diff = jd2 - jd1      ' diff immer positiv
  END IF
  IF diff > JDDeltaFromMinutes(60.01) THEN RETURN TRUE
  Return FALSE

End Function
```

Unsere Schicht endet um 16 Uhr 15 Minuten. Wir wollen wissen wie viele Minuten wir noch arbeiten müssen.

```
DIM zeit as DateAndTime
DIM ist, soll, count as Real
  zeit = SysGetTime()
  ist = JDFromDAT(zeit)       ' aktuelles Julianisches Datum

  zeit.hour = 20
  zeit.minute = 15
  zeit.second = 0
  soll = JDFromDAT(zeit)      ' Julianisches Datum vom Feierabend
```

```
IF ist > soll THEN
  count = (ist-soll)/JDDeltaFromMinutes(1)
  Print "Du hast schon seit "; count; " Minuten Feierabend!"
ELSE
  count = (soll-ist)/JDDeltaFromMinutes(1)
  Print "Du musst noch "; count; " Minuten arbeiten."
END IF
```

Interne Details. Diese Informationen können hilfreich sein, werden aber zur Arbeit mit dem Julianischen Datum nicht unbedingt benötigt.

- JDFromDAT und DATFromJD verwenden für Daten nach der Kalenderreform (d.h. ab dem 15.10.1582) den heute gültigen Gregorianischen Kalender. Dieser verwendet die erweiterte Schaltjahresregel, für Daten bis zum 04.10.1582 wird der julianische Kalender verwendet (einfache Schaltjahresregel).
- Historisch folgte in den damals führenden Ländern auf den 04.10.1582 sofort der 15.10.1582. Die dazwischen liegenden Daten existieren nicht. JDFromDAT und DATFromJD berücksichtigen das. Sie berücksichtigen aber nicht, dass viele Länder die Kalenderreform erst viel später durchführten.
- Historisch gab es vor Einführung des Julianischen Kalenders im Jahr 46 vor Christus keine Schaltjahre. Bei der Berechnung des Julianischen Datums wird das nicht berücksichtigt.
- Historisch existiert das Jahr Null nicht. Dem Jahr 1 vor Christus folgte sofort das Jahr 1 der christlichen Zeitrechnung. Bei kalendarischen Berechnungen wird das Jahr 1 vor Christus deshalb als Null gezählt, -1 entspricht dem Jahr 2 vor Christus usw.
- JDFromDAT ist tolerant gegenüber fehlerhaften Daten. Für den nicht existierenden 29.Februar 1999 wird beispielsweise das Julianische Datum des 1.März 1999 berechnet. Ein beliebiges Datum mit dem "nullten" des Monats (z.B. 0.3.1999) liefert das Julianische Datum des letzten Tags des Vormonats (im Beispiel den 28. Februar 1999).

- Die Nachkommastellen im Julianischen Datum entsprechen der Uhrzeit, wobei die Formel

  ```
  std/24 + min/1440 + sek/86400
  ```

  verwendet wird. Std, min und sek entsprechen dabei der Zeit, die seit 12 Uhr mittags vergangen ist. Für 14 Uhr gilt also: std = 2.
  Beispiele:

17.03.2016, 12 Uhr mittags	JD = 2457465
17.03.2016, 22 Uhr abends	JD = 2457465.41666667
18.03.2016, Null Uhr (morgens)	JD = 2457465.5
18.03.2016, 8 Uhr morgens	JD = 2457465.83333333

  Im Allgemeinen ist es nicht notwendig die Zahlenwerte zu kennen, wenn man mit dem Julianischen Datum arbeiten will.
- JDDeltaFromMinutes(x) entspricht entsprechend der obigen Formel dem Term x/1440. JDDeltaFromMinutes(24*60) liefert den Wert 1 (= 1 Tag).
- Die Funktionen DayOfWeek() und Weekday$() verwenden das Julianische Datum. Damit berücksichtigen sie automatisch den Gregorianischen und den Julianischen Kalender.

2.11.3 Speicherzugriff

Gelegentlich benötigt man einfach eine bestimmte Menge Speicher, die man nach den eigenen Vorstellungen organisieren kann. R-BASIC ermöglicht zwar keinen direkten Zugriff auf den Hauptspeicher des Computers (das wäre zu unsicher), stellt dem Programmierer aber 64 kByte "geschützten" (virtuellen) Speicher zur Verfügung, der aus Sicht des Programms als 64 kByte fortlaufender Speicher über die Adressen 0 bis 65635 (hexadezimal &h0 bis &hFFFF) angesprochen werden kann. Der Zugriff auf diesen Speicher erfolgt schreibend über diverse **Poke-** und lesend über die passenden **Peek**-Befehle (siehe Tabelle).

Befehl	Wirkung
POKE **<adr>**, **wert**	Schreibt ein Byte
DOKE **<adr>**, **wert**	Schreibt einen Integer- oder Word-Wert (2 Byte)
POKE$ **<adr>**, **<string>**	Schreibt eine String. Am Ende des Strings wird eine binäre Null als Ende-Kennung geschrieben. Es werden also LEN(<string>)+1 Bytes geschrieben.
SPOKE**<adr>**, **<struktur>**	Schreibt eine Struktur. Die Anzahl der geschriebenen Bytes hängt von der Struktur ab und beträgt sizeof(<struktur>) Bytes.
<nVar> = PEEK (**<adr>**)	Liest ein Byte
<nVar> = DEEK (**<adr>**)	Liest einen Word oder Integer-Wert (2 Byte).
<sVar$>** = PEEK$ (<adr>**)	Liest einen String. Das Stringende ist durch eine binäre Null gekennzeichnet. Es werden maximal 1024 Zeichen gelesen.
<stVar> = SPEEK (**<adr>**)	Liest eine Struktur. Die Anzahl der gelesenen Bytes hängt von der Struktur ab und beträgt sizeof(<stVar>) Bytes.
VPOKE <**adr**>, **wert**	Kompatibilitätsbefehl für KC-85 Kompatibilität. Beschreibt den Video-RAM des KC-85. Siehe Hinweise unten.
<nVar> = VPEEK (**<adr>**)	Kompatibilitätsbefehl für KC-85 Kompatibilität. Liest aus dem Video-RAM des KC-85. Siehe Hinweise unten.

<adr> numerischer Ausdruck, der einen Wert von 0 bis 65635 liefert. Liegt der Wert außerhalb dieses Bereichs, wird MOD 65636 gerechnet, d.h. es wird einfach wieder von vorne begonnen.
<string> Ein Stringausdruck.
<struktur> Eine Strukturvariable oder eine Funktion, die eine Struktur liefert. Es sind sowohl R-BASIC Strukturen als auch selbst definierte Strukturen zulässig.
<nVar> numerische Variable.
<sVar$> String-Variable.
<stVar> Strukturvariable.

Hinweise:

- R-BASIC verwaltet den Speicher in Blöcken zu 8 kByte, die erst angefordert werden, wenn sie benötigt werden.
- Dieser Speicherbereich wird auch an Libraries übergeben, die im PC/GEOS-SDK-Mode geschrieben sind. Außerdem wird er von einigen Objekten (z.B. BitmapContent) benutzt um große Datenmengen zu transferieren.
- Wollen Sie andere Datentypen (FILE, HANDLE, DWORD etc.) schreiben, müssen Sie sie in einer Struktur kapseln.
- R-BASIC kontrolliert nicht, ob die gelesenen Daten gültig sind, d.h. zum gelesenen Datentyp passen.
- Eine ausführliche Beschreibung der Befehle VPOKE und VPEEK finden Sie im Handbuch "Spezielle Themen", Vol. 3, Kapitel 18.3 (Kompatibilität mit dem KC-85-BASIC). Eine Beschreibung des KC-Video-RAM finden Sie im Anhang.

2.11.4 DATA-Zeilen

Die DATA-Anweisung dient dazu, feste Werte im Programm zu abzulegen, auf die zum gegebenen Zeitpunkt - bei Bedarf auch mehrfach - zugegriffen werden kann. Mit Hilfe der READ-Anweisung werden die Werte aus den DATA-Zeilen gelesen. RESTORE wird verwendet, um eine bestimmte DATA-Zeile anzuwählen. Dazu muss mit der Anweisung LABEL eine Codezeile markiert werden, auf die Restore verweisen kann.

DATA-Zeilen sind der klassische Weg um Daten (Zahlen und Strings) in einem BASIC-Programm unterzubringen. Die Verwendung von DATA-Zeilen ist ein veralteter Programmierstil und sie verbrauchen sehr viel Speicher im Programmcode. Für größere Datenmengen können Sie zum Beispiel externe Dateien verwenden, für Bilder können Sie auch die Picture-List (Menüpunkt: Extras) benutzen.
Manchmal, insbesondere wenn Sie nur kleine Datenmengen haben, sind DATA-Zeilen trotzdem sehr nützlich.

Sie können die Anweisungen DATA, READ, und RESTORE auch innerhalb einer Library verwenden. R-BASIC verwaltet die Werte für das Hauptprogramm und jede eingebundene Library getrennt, so dass eine gegenseitige Beeinflussung ausgeschlossen ist. Sie können also auch nicht vom Hauptprogramm aus mit READ auf die DATA-Werte einer Library zugreifen.

DATA

Syntax: **DATA** **Wert** [, **Wert** [, **Wert**] ...
Wert: jeweils eine Konstante vom Typ REAL oder STRING, Variablen sind nicht zulässig.

Beispiele:

```
DATA  12, 144, 13, 169
DATA  "Paul", "Müller", "Malocher"
```

READ

Syntax: **READ** **<var>** [, **<var>**] [, **<var>**] ...
<var> bezeichnet die zu belegenden Variablen.

Hinweise:

- Zulässig für <var> sind alle numerischen Datentypen sowie alle String-Typen. Dazu zählen auch Feld- und Struktur-Elemente.
- Der Typ der Variablen muss kompatibel zum jeweiligen Wert in der DATA-Zeile sein, sonst kommt es zu einem Laufzeitfehler und das Programm wird beendet.

Beispiele:
Der folgende Code liest die Werte aus den DATA-Zeilen aus dem Beispiel zum Befehl DATA (oben).

```
READ  A,  B,  C,  D
READ  Vorname$,  Name$,  Job$
```

RESTORE

Syntax: **RESTORE**
RESTORE <label>
<label> ist ein im Programm definiertes Label.

Hinweise:
- Wird kein Parameter (<label>) angegeben, so wird die erste DATA-Zeile des Programms angewählt.
- Existiert das Label nicht, kommt es zu einem Compilerfehler.
- Weist das Label nicht direkt auf eine DATA-Zeile, so wird die nächste im Programm vorkommende DATA-Zeile angewählt.
- Weist das Label hinter die letzte DATA-Zeile, so kommt es bei der nächsten READ-Anweisung zu einem Laufzeitfehler

Beispiele:

```
  DATA 1,2,3,4,5,6,7,8
LABEL Liste1
  DATA "a","b","c","d","e","f"
....
  RESTORE    ' zur ersten DATA-Zeile
  RESTORE   Liste1    ' zur DATA-Zeile ab Liste1
```

Allgemeine Hinweise:
- Sind mehr Werte in einer DATA-Zeile, als Variablen gelesen werden, so setzt die nächste READ-Anweisung mit dem nächsten Wert in dieser Zeile fort.
- Sollen mehr Werte gelesen werden, als Werte in einer DATA-Zeile sind, so wird mit der nächsten DATA-Zeile fortgesetzt.
- Stößt der Interpreter beim Programmablauf auf eine DATA-Zeile, wird diese übersprungen.
- Versucht das Programm mehr Werte zu lesen, als durch DATA-Zeilen definiert sind, kommt es zu einem Laufzeitfehler und das Programm wird beendet.

Tipps:
- Verwenden Sie in einer DATA-Zeile nach Möglichkeit nur einen Typ von Werten (z.B. nur Zahlen oder nur Strings)
- Fassen Sie die DATA-Zeilen zu einem Block im Programm zusammen. Es bietet sich hier das DIM & DATA-Fenster an.

Beispiel:

```
LABEL Liste1
  DATA 10, 12, 19, 20
  DATA 10.8, 7.06, 51.2, 8.13
          ' Beachten: Punkt als Dezimaltrennzeichen

RESTORE Liste1      ' DATA-Zeiger setzen
READ a, b, c, d     ' liest die Werte 10, 12, 19, 20
READ e, f, g, h     ' liest die Werte 10.8, 7.06, 51.2, 8.13
....
RESTORE Liste1
READ e, f, g, h     ' liest wieder die Werte 10, 12, 19, 20
```

Kompatibilität
R-BASIC unterstützt auch die in vielen BASIC-Interpretern verwendete Kombination "RESTORE Zeilennummer". Das kann die Übertragung fremder BASIC-Programme vereinfachen. Die "Zeilennummer" muss dabei explizit angegeben sein (z.B. "1000 DATA ..."). Sie sollten diese Variante in eigenen Programmen nicht verwenden.

LABEL

Die Anweisung LABEL (Marke) vereinbart ein Ansprungziel für GOTO, RESTORE oder GOTO.

Syntax: **LABEL** **<sprungZiel>**
<sprungZiel>: Name, unter dem die Stelle erreicht werden kann.

Beispiel:

```
GOTO  keinFehler    ' verzweigt das Programm nach unten
....   ' hier passiert etwas anderes
LABEL  keinFehler
...    ' hier geht es dann weiter
```

Hinweise:
- Ein Sprung-Ziel (Label) muss noch nicht definiert sein, bevor es das erste Mal verwendet wird. Eine Verwendung definiert das Label vorläufig. So auch im Beispiel oben.
- Wird ein Sprung-Ziel verwendet, ohne es später mit LABEL endgültig zu definieren, kommt es zu einem Compilerfehler.
- Die Verwendung von GOSUB ist veraltet und wird nur noch aus Kompatibilitätsgründen unterstützt. Sie sollten GOSUB in eigenen Programmen nicht verwenden.

2.11.5 System Services

Hier finden Sie eine Liste der wichtigsten Systemservices, die von R-BASIC aus benutzt werden können, sowie Informationen, wo sie beschrieben werden. Die meisten dieser Services werden von GEOS-Objekten bereitgestellt.

Tastatur

Die meisten Objekte unterstützen die Tastatur automatisch, ohne weiteres Zutun des Programmierers. Zum direkten Zugriff auf die Tastatur stehen Ihnen folgende Möglichkeiten zur Verfügung:

- Für einfache Fälle stehen Ihnen die Funktionen **InKey$**, **GetKey**, **GetKeyLP** und **GetKeyState** zur Verfügung, die im Kapitel 2.7.2 des Programmierhandbuchs beschrieben werden.
- Sie können einen **Tastaturhandler** schreiben. Tastaturhandler werden automatisch aufgerufen, wenn der Nutzer eine Taste betätigt. Sie werden im Handbuch "Spezielle Themen", Kapitel 14 (Arbeit mit der Tastatur) beschrieben.

Maus

Die meisten Objekte unterstützen die Maus automatisch, ohne weiteres Zutun des Programmierers. Wenn Sie selbst direkt auf Mausereignisse reagieren wollen müssen Sie einen **Maushandler** schreiben. Maushandler werden automatisch aufgerufen, wenn der Nutzer die Maus bewegt oder eine der Maustasten betätigt. Sie werden im Handbuch "Spezielle Themen", Kapitel 17 (Arbeit mit der Maus) beschrieben.

Drucken

Um aus einem R-BASIC Programm heraus drucken zu können müssen Sie ein Objekt der Klasse **PrintControl** einbinden. Dieses Objekt wird im Objekt-Handbuch, Kapitel 4.14, beschrieben. Eventuell benötigen Sie noch ein Objekt der Klasse **PageSizeControl**. Dieses Objekt wird im Kapitel 4.15 des Objekt-Handbuchs beschrieben.

Clipboard

Über die Zwischenablage (Clipboard) können R-BASIC Programme Daten (z.B. Texte oder Grafiken) mit anderen Programmen austauschen. Die Arbeit mit der Zwischenablage wird im Handbuch "Spezielle Themen", Kapitel 5 (Arbeit mit der Zwischenablage) beschrieben.

Hilfedateien

R-BASIC stellt Ihnen das GEOS-weite Hilfesystem zur Verfügung. Im Kapitel 4 des Handbuchs "Spezielle Themen" (Einbinden von Hilfedateien) ist beschrieben, wie man das Hilfesystem einsetzt.

Timer

Ein Timer stellt eine Routine bereit, die in regelmäßigen Abständen automatisch aufgerufen wird. Timer sind im Kapitel 16 des Handbuchs "Spezielle Themen" (Timer) ist beschrieben.

Dateiarbeit

R-BASIC unterstützt die Arbeit mit DOS- und GEOS-Dateien. Sie können sowohl Binär- als auch Text-Dateien lesen und schreiben. Die Arbeit mit Dateien ist in folgenden Kapiteln des Handbuchs "Spezielle Themen" beschrieben.

- Kapitel 6: Das Dateisystem
- Kapitel 7: Arbeit mit Pfaden und Ordnern
- Kapitel 8: Verwaltung von Dateien
- Kapitel 9: Arbeit mit Dateien
- Kapitel 10: Arbeit mit Laufwerken und Datenträgern

Um in den Genuss der Vorteile von GEOS-VM-Dateien zu kommen müssen Sie die Library "VMFiles" einbinden. Diese Library muss separat von der R-BASIC Webseite heruntergeladen werden.

Dokumente

Im Kapitel 15 des Handbuchs "Spezielle Themen" (Implementieren eines Dokument-Interfaces) finden Sie eine ausführliche Anleitung, wie man unter R-BASIC die Funktionen des "Datei" Menüs implementiert. Einen großen Teil der Funktionen übernimmt dabei ein Objekt der Klasse **DocumentGuardian**. Diese Objekte arbeiten mit der Library "DocumentTools" zusammen und sind im Objekt-Handbuch, Kapitel 4.13, beschrieben.

Automatisches Geometriemanagement

Eine der wesentlichen Eigenschaften des GEOS Systems ist seine Fähigkeit, die Objekte automatisch so anzuordnen, dass sie alle gut sichtbar sind ohne sich zu überlappen. Diese Fähigkeit ist auf elementarer Systemebene implementiert und im Objekt-Handbuch, Kapitel 3.3 (Geometriemanagement) beschrieben.

2.12 Verwendung von Libraries

2.12.1 Konzeptionelles

Libraries (deutsch: Bibliotheken) sind Sammlungen von Funktionen, SUBs, Konstanten und Struktur-Typen, die von anderen Programmen und Libraries verwendet werden können. Libraries können keine statischen UI-Objekte enthalten, aber UI-Objekte können zur Laufzeit erzeugt werden (siehe unten).

Libraries bieten folgende Vorteile:
- Verwendung von geprüftem Code. Die Routinen einer Library sind meist sehr gründlich getestet. Bugfixes in einer Library wirken sich sofort auf alle Programme aus, die diese Library verwenden, ohne dass das Programm neu compiliert werden muss.
- Schneller Programmentwicklung. Routinen aus einer Library muss man nicht selbst oder noch einmal schreiben.
- Mehrere BASIC-Programme können die gleiche Library gleichzeitig verwenden. Sie können Library-Routinen also so verwenden, als gehörten sie ausschließlich zu Ihrem Programm.

Sie können eine Library für R-BASIC sowohl mit R-BASIC als auch mit dem PC/GEOS SDK schreiben. Eine mit dem SDK geschriebene Library kann R-BASIC um Funktionen erweitern, die im ursprünglichen Konzept nicht vorgesehen sind. Um eine R-BASIC-Library mit dem SDK zu schreiben müssen Sie das "SDK Library Kit" von der R-BASIC Webseite herunterladen und der dort enthaltenen Anleitung folgen.

Um eine Library mit R-BASIC zu schreiben, gehen Sie folgendermaßen vor:
1. Öffnen Sie ein neues R-BASIC Programm
2. Wählen Sie aus dem Menü "Extras" den Punkt "In BASIC Library umwandeln". Es wird ein neues Codefenster "Exports" aktiv. Das Fenster "UI-Objekte" wird deaktiviert, weil R-BASIC Libraries keine statischen UI-Objekte enthalten können.
3. Speichern Sie die Library unter einem aussagekräftigen Namen. Dieser Name wird später zum Einbinden der Library verwendet. Der Ort, an dem Sie die Library speichern, spielt keine Rolle.
4. Schreiben Sie den BASIC Code genau so als ob Sie ein Programm erstellen. Vereinbarungen (Anweisungen: DIM, DECL, CONST, STRUCT), die von der Library für andere Programme bereitgestellt werden sollen, müssen in das "Exports" Fenster. Library-interne Vereinbarungen, die nicht exportiert (für andere Programme bereitgestellt werden) werden sollen, sollten Sie im DIM & DATA-Fenster unterbringen.
5. Compilieren Sie Ihre Library. R-BASIC speichert den compilierten Code im Ordner Userdata\R-BASIC\Library\Bin. Dort können ihn alle BASIC-Programme finden.

Um den Librarycode zu testen müssen Sie
- die Library compilieren
- ein Programm schreiben, das die Library einbindet (Anweisung: Include "Name der Library") und dort die Library-Routinen aufrufen.

Include

Die Anweisung Include bindet eine Library ein. Jedes Programm kann bis zu 32 Libraries einbinden, wobei auch die Libraries zählen, die von eingebundenen Libraries verwendet werden.
Include erwartet den GEOS Namen der Library, das heißt, die Groß- und Kleinschreibung als auch eventuelle Leerzeichen im Namen müssen beachtet werden.

Syntax: **INCLUDE "LibraryName"**

Beispiel:

```
INCLUDE  "Demo Library"
```

customError

Die numerische Systemvariable **customError** (Datentyp INTEGER) kann benutzt werden um eine selbstdefinierte Fehlernummer zu speichern. Sie kann geschrieben und gelesen werden. Das ist insbesondere für Libraries, auch für SDK-Libraries, interessant. R-BASIC selber verwendet diesen Wert nicht.

Beispiel

```
  customError = 12
```

```
  IF customError = 4 THEN  Print "Fehler 4 aufgetreten."
```

Beispiel:
Wir wollen eine Library schreiben, die 2 Routinen und zwei Konstanten exportiert. Die Funktion Schummel soll eine Zahl von 1 bis 6 liefern, wobei die 6 aber häufiger vorkommt. Die Sub Meldung soll in Abhängigkeit von einer Zahl, die der Routine übergeben wird, zwei unterschiedliche Meldungen ausgeben. Dazu brauchen wir die beiden Konstanten.
Den kompletten Code für dieses Beispiel finden Sie im Ordner "Beispiel\Library", Dateien "Library einfach" und "Library einfach Test Programm".

Im Exports-Fenster vereinbaren wir:

```
CONST    STATE_WIN  = 1
CONST    STATE_LOSE = 2
DECL     FUNCTION Schummel() AS Real
DECL     SUB Meldung(state as Real)
```

Den Code für die beiden Routinen schreiben wir im BASIC-Code Fenster:

```
FUNCTION Schummel() AS Real
DIM x

  x = 10*Rnd()   ' 0 ... 9.9999999
  x = Int(x)+1   ' 1 ... 10
  IF  x > 6 THEN x = 6
  Return x
END Function

SUB Meldung(state as Real)
  IF state = STATE_WIN THEN
    MsgBox "Wow, du hast gewonnen. " \
        + "Das hätte ich dir gar nicht zugetraut."
  ELSE
    WarningBox "Du hast ja so mies gespielt! " \
        + "Kein Wunder, dass du verloren hast."
  END IF
End SUB
```

Wir speichern die Library unter einem aussagekräftigen Namen, z.B. "Test Lib" und compilieren sie dann. Der Name wird für die Include-Anweisung benötigt.

Unser Testprogramm könnte so aussehen:

```
Include "Test Lib"
ClassicCode
DIM x
x = schummel()
Print x
IF x > 4 THEN
  Meldung(state_win)
ELSE
  Meldung(state_lose)
End IF
```

Die Anweisung ClassicCode bewirkt, dass R-BASIC automatisch einige UI-Objekte anlegt (ein Primary mit einem View und einem BitmapContent), so dass die Print-Anweisung verwendbar ist und außerdem der Code beim Programmstart automatisch abgearbeitet wird (sogenannter klassischer BASIC Modus).

Wenn wir jetzt eine Änderung an der Library in der Library vornehmen müssen wir sie neu compilieren. Beim nächsten Start unseres Testprogramms wird die geänderte Library verwendet. Es ist dazu nicht nötig (aber trotzdem sinnvoll), die Library zu speichern.

Beim Schreiben einer Library können Sie alles tun, was sie auch beim Schreiben eines normalen Programms tun können. Es ist aber ist folgendes zu beachten:

- Eine Library kann nicht "gestartet" werden. Es gibt kein "Hauptprogramm", nur eine Sammlung von Unterprogrammen.
- Bezeichner, die von der Library anderen Programmen zur Verfügung gestellt werden sollen (man sagt: sie sollen "exportiert" werden), müssen im EXPORTS-Fenster vereinbart werden.
- Exportiert werden können:
  - SUB's
  - FUNCTION's
  - ACTION-Handler
  - STRUCT's
  - Konstanten (CONST-Anweisung)
  - Variablen aller Typen (DIM-Anweisung). Diese Variablen werden vom Programm, dass die Library benutzt, als globale Variablen verwendet. Sie können zum Beispiel benutzt werden um Daten zwischen dem Programm und der Library auszutauschen.
    Hinweis: Globale Variablen einer Library (egal ob exportiert oder nicht) werden im globalen Variablenspeicher des aufrufenden Programms abgelegt. Wenn mehrere Programme eine Library gleichzeitig verwenden, so wird für jedes Programm ein eigener Satz dieser Variablen angelegt. Library und Programm verwenden diese Variablen so, als ob sie allein im System sind. Eine gegenseitige Beeinflussung verschiedener Programme ist ausgeschlossen.
- Nicht exportiert werden können LABEL's.
- Libraries können keine UI-Objekte enthalten. Das UI-Objekte Fenster ist gesperrt. Weiter unten ist beschrieben, wie Sie trotzdem Objekte in Libraries verwenden können.
- Eine Library kann eine eigene PictureList haben (siehe Kapitel 2.8.6.2: Verwendung der "Picture-List"). Auf diese PictureList kann nur innerhalb der Library zugegriffen werden. Wenn Sie Bilder aus einer Library-PictureList verwenden wollen müssen Sie eine Library-Routine schreiben, die auf die PictureList der Library zugreift.
- DATA Zeilen innerhalb einer Library sind zulässig. Auch hier gilt: Auf diese DATA-Zeilen kann nur von innerhalb der Library zugegriffen werden. DATA Zeilen sollten deshalb im DIM & DATA Fenster stehen.

Wichtig!

Wenn ein Programm eine Routine (Sub, Function, Actionhandler) aus einer Library verwendet so wird die Routine über eine Nummer identifiziert. Diese Nummer entspricht der Position in der Liste der exportierten Routinen. Dabei zählen nur Routinen. Vereinfacht gesagt zählt R-BASIC die DECL-Anweisungen im EXPORTS Fenster. Das bedeutet konkret:

- Wenn Sie die Reihenfolge der exportierten Routinen ändern wird die Library inkompatibel. Alle Programme, die diese Library benutzen müssen dann neu compiliert werden.
- Wenn Sie neue Routinen **vor** den bereits vorhandenen Routinen einfügen wird die Library ebenfalls inkompatibel. Sie dürfen neue Routinen im EXPORT-Fenster nur **nach** allen bereits vorhandenen Routinen einfügen.

- Für den Export von Structs, Konstanten und Variablen gelten diese Einschränkungen nicht. Hier können Sie beliebig schieben oder hinzufügen.
- Die genannten Einschränkungen gelten nur für exportierte Routinen. Routinen, die im DIM & DATA Fenster deklariert sind, unterliegen diesen Einschränkungen nicht.

Natürlich wird eine Library auch inkompatibel, wenn Sie die Parameter einer Routine ändern, Konstanten einen anderen Wert geben oder dergleichen.

Beispiel. Das ursprüngliche EXPORT-Fenster sieht so aus:

```
CONST  ZAHL_1 = 12
CONST  TEXT_1 = "Hallo Welt"
DECL   SUB Routine1 ( )
DECL   FUNCTION Routine2(x as REAL) as REAL
```

Inkompatible Änderung: Reihenfolge vertauscht

```
CONST  ZAHL_1 = 12
CONST  TEXT_1 = "Hallo Welt"
DECL   FUNCTION Routine2(x as REAL) as REAL
DECL   SUB Routine1 ( )
```

Inkompatible Änderung: Neue Routine eingeschoben

```
CONST  ZAHL_1 = 12
CONST  TEXT_1 = "Hallo Welt"
DECL   SUB Routine3 (z as INTEGER )
DECL   SUB Routine1 ( )
DECL   FUNCTION Routine2(x as REAL) as REAL
```

Kompatible Änderung: Neue Routine angefügt

```
CONST  ZAHL_1 = 12
CONST  TEXT_1 = "Hallo Welt"
DECL   SUB Routine1 ( )
DECL   FUNCTION Routine2(x as REAL) as REAL
DECL   SUB Routine3 (z as INTEGER )
```

Kompatible Änderungen: Konstanten hinzugefügt und verschoben

```
CONST  ZAHL_1 = 12
CONST  ZAHL_2 = 559.8

DECL   SUB Routine1 ( )
DECL   FUNCTION Routine2(x as REAL) as REAL
CONST  X_1 =    77
CONST  X_2 =    85.7

DECL   SUB Routine3 (z as INTEGER )
CONST  TEXT_1 = "Hallo Welt"              ' verschieben ist OK
CONST  TEXT_2 = "Hallo Jungs"
```

Die Library-Versionsnummer

Jedes Mal, wenn eine neue Version einer Library veröffentlicht wird sollten Sie die Versionsnummer ändern.
Die Werte für Major und Minor können im Menüpunkt "Programm" geändert werden. Der Major-Wert sollte geändert (vergrößert) werden, wenn wesentliche neue Funktionen dazugekommen sind. Ein neuer Wert für Minor sollte Bugfixes oder kleinere Änderungen anzeigen. Insbesondere wenn Sie neue Routinen hinzugefügt haben sollen Sie die Versionsnummer (Major oder Minor) vergrößern.
Programme werden mit neueren Versionen einer Library problemlos zusammenarbeiten, wenn Sie die oben angegebenen Hinweise zur Kompatibilität beachten. Eine ausführliche Beschreibung der Versionsnummern finden Sie im Kapitel 2.11.1.

ConvertObjForSDK

Dies ist eine Hilfsfunktion für Programmierer von SDK Libraries. "Normale" Objektvariablen speichern R-BASIC Objekte in einer Art und Weise, mit der man im SDK nichts anfangen kann. Die Funktion **ConvertObjForSDK** liefert eine "Objektvariable", die so angepasst wurde, dass man aus einer SDK-Routine heraus mit dem BASIC-Objekt kommunizieren kann.

Syntax: **<objVar>** = ConvertObjForSDK (**<objExpression>**)

Wichtige Hinweise:

- Verwenden Sie **ConvertObjForSDK** nur, wenn Sie eine Routine aus einer SDK-Library aufrufen wollen **und** der Programmierer der SDK-Library hat explizit festgelegt, dass **ConvertObjForSDK** zu verwenden ist!
- Nutzen Sie den von ConvertObjForSDK zurückgegebene Wert **niemals** für eine "normale" Objekt-Routine von R-BASIC, oder GEOS wird crashen!

Technische Details:
R-BASIC Objektvariablen enthalten eine Referenz auf ein Objekt, die nicht identisch mit dem im SDK verwendeten Objekt-Pointer (optr) des Objekts ist. ConvertObjForSDK ersetzt diese BASIC interne Referenz durch den realen optr des Objekts. Weitere Details zu Objektvariablen und wie man ConvertObjForSDK einsetzt finden Sie in der Beschreibung, wie man SDK Libraries erstellt. Diese kann auf der R-BASIC Homepage heruntergeladen werden.

2.12.2 UI-Objekte in Libraries

Dieses Kapitel richtet sich an fortgeschrittene Programmierer.

Wie bereits oben erwähnt können Sie in einer Library keinen Objekt-Tree im UI Codefenster anlegen. Es ist jedoch zulässig Objekte zur Laufzeit des Programms anzulegen und später wieder zu vernichten. Die dazu benötigten Befehle CreateObject und DestroyObject sind im Kapitel 2.1.5 (Anlegen und Vernichten von Objekten zur Laufzeit) des Objekthandbuchs beschrieben. Außerdem sollten Sie die Kapitel 2.1.3 (Verwaltung von Objektblöcken, Befehle CreateObjBlock und DestroyObjBlock) sowie 3.3.8 (Hintertürchen für Programmierer, Befehle ObjAddHint und ObjRemoveHint) aus dem Objekthandbuch lesen.

Um eine Library zu schreiben, die Objekte, z.B. eine Dialogbox, bereitstellt, sollten Sie folgendermaßen vorgehen:

- Schreiben Sie in der Library eine Initialisierungs-Routine (Function mit dem Rückgabetyp Object), die den Objekt-Tree anlegt.
  Die Initialisierungs-Routine sollte das Top-Objekt des gerade erzeugten Objekt-Trees zurückgeben.
- Schreiben Sie in der Library eine Cleanup-Routine, die von der Library erzeugten Tree vernichtet und den Objektblock wieder freigibt.
- Sollte es notwendig sein, auf andere Objekte als das Top-Objekt des Objekt-Trees zuzugreifen - was üblicherweise der Fall ist - so sollten Sie Routinen in der Library schreiben, die
  - entweder das gewünschte Objekt zurückgeben, so dass das Programm selbst auf das Objekt zugriefen kann
  - oder die auf das UI-Objekt zugreifen und die gewünschten Daten zurückgeben bzw. setzen.
- Falls Sie mehrere Objekt-Trees in der Library haben wollen (z.B. mehrere Dialogboxen) sollten Sie auch mehrere Initialisierungs- und Cleanup-Routinen schreiben.

Im Programm, dass die Library nutzt, müssen Sie folgendes tun:

- Rufen Sie die Initialisierungs-Routine der Library im OnStartup-Handler ihres Programms auf. Nur falls Sie die Library-UI sofort benötigen sollten Sie den OnInit-Handler verwenden. Binden Sie das von der Initialisierungs-Routine zurückgegebene Objekt in den Objekt-Tree Ihres Programms ein, indem Sie die Instancevariable "parent" dieses Objekts belegen.
  Sie sollten die Initialisierungs-Routine der Library nicht rufen, wenn GEOS nach einem Systemneustart mit geöffnetem Programm wieder hochfährt. Beachten Sie dazu das Beispiel unten.
- Rufen Sie die CleanUp-Routine im OnExit-Handler ihres Programms.
  Sie sollten die Cleanup-Routine der Library nicht rufen, wenn GEOS mit geöffnetem Programm herunterfährt. Beachten Sie dazu das Beispiel unten.

Beispiel

Die folgenden Codefragmente sind an die Beispiele "UI Lib" und "Dialog Library" sowie den dazugehörigen Testprogrammen angelehnt. Die vollständigen Dateien finden Sie im Ordner "Beispiel\Library". Die beiden Beispiele beleuchten unterschiedliche Aspekte des Themas.

Erzeugen eines UI-Trees

- Verwenden Sie CreateObjBlock um einen neuen Objektblock anzulegen und CreateObject um die gewünschten Objekte darin zu erzeugen.
- Setzen Sie die Instance-Variablen der Objekte auf die Werte, die Sie sonst im UI-Code setzen würden. Verwenden Sie den Befehl ObjAddHint, um Instancevariablen zu setzen, die sonst nur im UI-Code verfügbar sind. Vergessen Sie nicht, die Objekte in einem Tree zu verlinken (Instancevariable parent setzen, Children ist read-only).

Die Initialisierungs-Routine. Sie soll folgenden UI-Tree nachbilden:

```
Dialog LibraryDialog
    Caption$ = "Notizen eingeben"
    Children = DialogClearButton
  End Object

Button DialogClearButton
    Caption$ = "Text löschen"
    ActionHandler = DoClearText
End Object
```

Dabei muss der Actionhandler "DoClearText" irgendwo in der Library implementiert sein.

```
FUNCTION BuildLibUI () AS object
dim objBlock as handle
dim dlg, obj as object

    ! Anlegen des Objektblocks, der die Objekte aufnehmen soll
  objBlock = CreateObjBlock()

    ! In diesem Objektblock erzeugen wir die UI-Objekte
    ! und setzen die entsprechenden Instancevariablen
  dlg = CreateObject (objBlock, Dialog)
  dlg.caption$ = "Notizen eingeben"

  ob = CreateObject (objBlock, Button)
  ob.caption$ = "Text löschen"
  ob.actionHandler = DoClearText
  ob.parent = dlg, 0                  ' Button in Tree einbinden
```

Zugriff auf Objekte in der Library

Nur die Library selbst kennt die genaue Struktur des UI-Trees. Deshalb sollten Sie das "Finden" von Objekten als Routine in der Library programmieren. Falls Sie später die Struktur des UI-Trees ändern, müssen Sie nur Code in der Library ändern. Die Programme, welche die Library nutzen, brauchen dann nicht neu compiliert werden.

Die folgende Routine liefert den Button, der oben, in der Routine BuildLibUI, angelegt wurde. Ihr wird das Top-Objekt des Library-Trees übergeben. Dieses ist das einzige, dass das aufrufende Programm wirklich kennt.

```
FUNCTION LibFindButton (topObj as OBJECT) AS OBJECT
  ' Wir wissen: Der Button ist das erste Child des Dialog-Objekts
  return topObj.Children(0)
END FUNCTION
```

Im Beispielcode der Library "UI Lib" wird erklärt, wie man von einem ActionHandler aus ein Objekt im UI-Tree der Library finden kann.

Vernichten des UI-Trees

Um das System beim Beenden des Programms sauber zu halten sollten Sie dafür sorgen, dass alle Objekte und Objektblöcke, die Sie angelegt haben, auch wieder vernichtet werden. Die folgende Routine erledigt das für einen kompletten Tree. Ihr wird das Top-Objekt des zu vernichtenden Trees übergeben. Anschließend müssen Sie noch den leeren Objektblock freigeben. Den Code dafür finden Sie in den Beispielen.

Die Routine DestroyTree arbeitet rekursiv. Das heißt, sie ruft sich selbst auf. Rekursive Programmierung ist schwer zu verstehen und schwer zu erklären. Aber sie ist sehr leistungsfähig und der übliche Weg, Baumstrukturen durchzugehen. Sie können diese Routine einfach benutzen. Eine kurze Erklärung finden Sie im Code der Beispiele.

```
SUB DestroyTree (obj as object)
dim ch as object

  WHILE obj.numChildren       ' noch children da?
    ch = obj.children(0)      ' erstes Child holen
    DestroyTree(ch)           ' ... und samt seiner
                              ' Unter-Children vernichten
                              ' das ehemals zweite Child ist
                              ' jetzt das erste Child
    WEND

    ' Das Objekt hat jetzt keine Children mehr
  obj.parent = NullObj(),0    ' Objekt aus dem Tree nehmen
  DestroyObject(obj)          ' Objekt vernichten

END SUB
```

Beachten Sie, dass diese Beispielroutine nur den regulären Tree vernichtet. Sie prüft nicht ab, ob das zu vernichtende Objekt ein View ist, das ein Content hat. Bei Bedarf können Sie vor dem eigentlichen Vernichten des Objekts die Instance-variable Class$ abfragen. Sie enthält die Objektklasse im Klartext (z.B. "VIEW") und Sie können dann das Content oder den damit verbundenen VisTree vernichten. Die Routine DestroyTree arbeitet auch für den VisTree.

Verwendung im Programm

Im Programm müssen Sie den Library UI-Tree anlegen (am Programmstart) und wieder vernichten (am Programmende). Dazu verwenden Sie die OnStartup und OnExit Handler des Application-Objekts. Die einfachste Variante sieht so aus:

```
SYSTEMACTION AppStartup
DIM ob as OBJECT
    ob = BuildLibUI()
    ob.parent = DemoPrimary, 0     ' Einbinden in den GenTree
END ACTION
```

```
SYSTEMACTION AppExit
DIM libTopObj as Object
    libTopObj = DemoPrimary.Children(0)
    DestroyLibUI(libTopObj)
END ACTION
```

Jedes Mal, wenn das Programm startet, wird die Library UI erzeugt und bei jedem Programmende wird sie vernichtet. Der wesentliche Nachteil bei diesem einfachen Vorgehen ist, dass das Vernichten und Erzeugen auch passiert, wenn GEOS bei offenem Programm herunterfährt. Das kostet nicht nur Zeit, sondern es gehen auch alle Eingaben (z.B. Texte oder Auswahlfelder), die in der Library UI gemacht wurden, verloren.

Deswegen ist es eine bessere Idee, abzufragen ob GEOS herunterfährt oder nach einem System-Shutdown neu startet. Der Parameter "flags" enthält spezielle Bits, die genau das anzeigen. Das Flag AF_RESTORE ist gesetzt, wenn das Programm nach einem System Shutdown wieder öffnet. Das Flag AF_SHUTDOWN ist gesetzt, wenn das System herunterfährt, unser Programm aber noch offen ist. Um zu testen, ob das Flag gesetzt ist, verwenden wir die logische Operation AND. Ergibt diese Operation den Wert Null, so ist das Bit nicht gesetzt und wir müssen handeln.

```
SYSTEMACTION AppStartup
dim ob as object
  IF (flags AND AF_RESTORE) = 0 THEN
    ' Das Programm startet neu.
    ' Wir müssen die Library-UI erzeugen.
    ob = BuildLibUI()
    ob.parent = DemoPrimary, 0     ' Einbinden in den GenTree
  END IF
END ACTION
```

```
SYSTEMACTION AppExit
DIM libTopObj as Object
  IF (flags and AF_SHUTDOWN) = 0 THEN
    ' Das Programm wird geschlossen.
    ' Wir müssen die Library-UI vernichten
    libTopObj = DemoPrimary.Children(0)
    DestroyLibUI(libTopObj)                ' Cleanup-Routine
    END IF
END ACTION
```

Es ist möglich, die UI-Objekte der Library in globalen Variablen zwischenzuspeichern. Das erleichtert den Zugriff auf sie. Sie müssen aber beachten, dass globale Variablen einen Systemneustart nicht überleben. Im Beispielcode des Programms "Dialog Library Test Programm" ist beschrieben, wie Sie in diesem Fall vorgehen können.

(Leerseite)