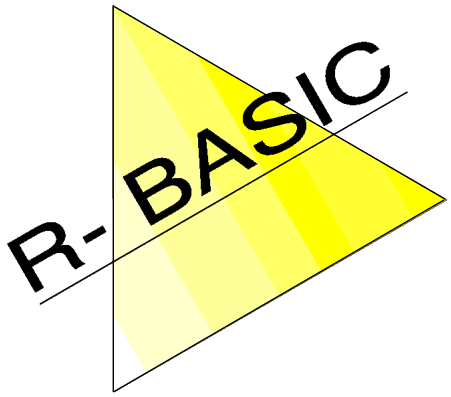
R-BASIC

Einfach unter PC/GEOS programmieren



Programmierhandbuch

Volume 3
Unterprogramme,
Eingaben durch den Nutzer, Grafik

Version 1.0

(Leerseite)

Inhaltsverzeichnis

(Leerseite)

2.6 Unterprogramme

Einem Programm eine übersichtliche Struktur zu geben ist eine wesentliche Voraussetzung, um verzwickte und schwer zu findende Fehler zu vermeiden.

Unterprogramme (Sub-Routinen) sind in sich geschlossene Programmabschnitte, quasi Programme innerhalb eines Programms. Die Vorteile bei der Verwendung von Unterprogrammen sind:

- Strukturierung: Das Programm wird wesentlich besser lesbar, da es in kleine, voneinander unabhängige Einheiten (Unterprogramme) zerlegt wird.
- Stabilität: Kleine Unterprogramme sind wesentlich einfacher fehlerfrei zu halten als ein komplexes Riesenprogramm.
- Kapselung: Die Verwendung lokaler Variablen garantiert, das sich die Programmteile nicht unerwünscht gegenseitig beeinflussen.
- Mehrfache Verwendbarkeit: Unterprogramme können so oft gerufen werden, wie es nötig ist. Unterprogramme können andere Unterprogramme aufrufen, sie können sich sogar selbst aufrufen.

R-BASIC unterstützt folgende Arten von Unterprogrammen:

- Unterprogramme ohne Rückgabe von Funktionswerten (SUB)
  SUB's werden über ihren Namen aufgerufen.

```
NameDerSub [ < ParameterListe> ]
```

- Unterprogramme mit Rückgabe von Funktionswerten (FUNCTION)
  Funktionen werden ebenfalls über ihren Namen aufgerufen. In den meisten Fällen stehen Funktionen auf der rechten Seite einer Zuweisung.

```
<variable> = NameDerFunction ( [ <Paramterliste> ] )
```

- Action-Handler für Objekte
  Actionhandler werden automatisch von ihren Objekten aufgerufen. Sie können nicht von anderen Teilen des Programms aus gerufen werden.
- Aus Kompatibilitätsgründen wird die Kombination GOSUB / RETURN unterstützt. Sie sollten GOSUB in eigenen Programmen nicht verwenden.

Konzept: Lokale Variablen

Ein Unterprogramm hat vollen Zugriff auf alle global definierten Variablen, Konstanten, Strukturen usw. und kann mit diesen arbeiten. Globale Variablen werden üblicherweise im DIM & DATA Fenster vereinbart. Wenn aber alle Unterprogramme ausschließlich mit globalen Variablen arbeiten wird das schnell unübersichtlich und es kann zu einer unerwünschten gegenseitigen Beeinflussung der Programmteil führen. Deswegen kann man Variablen "lokal", das heißt nur für dieses eine Unterprogramm definieren. Dazu schreibt man die entsprechende DIM-Anweisung innerhalb des Unterprogramms. Diese Variablen sind dem Compiler nur innerhalb des Unterprogramms bekannt und können auch nur innerhalb dieses Unterprogramms benutzt werden.

Stößt der Compiler innerhalb eines Unterprogramms auf eine Variable (z.B. A$), sucht er zuerst, ob diese lokal definiert ist. Ist das der Fall, wird die lokale Variable verwendet. Nur wenn sie nicht lokal definiert ist, wird die entsprechende global definierte Variable verwendet. Diese Technik nennt man Kapselung. Das hat drei sehr praktische Folgen:

- Man kann die lokalen Variablen unabhängig von den global definierten Variablen, Konstanten usw. benennen. Bei Namensgleichheit wird auf jeden Fall die im Unterprogramm definierte Variable verwendet.
- Verschiedene Unterprogramme brauchen ebenfalls keine Rücksicht aufeinander nehmen, auch dann nicht, wenn sie sich gegenseitig aufrufen. Damit kann man zum Beispiel ein Unterprogramm aus einem anderen Programm herüberkopieren und braucht sich keine Sorgen um die Benennung der lokalen Variablen machen.
- Ein Unterprogramm hat nur Zugriff auf seine eigenen lokalen Variablen und auf die globalen Variablen. Ein Unterprogramm hat **keinen** Zugriff auf die lokalen Variablen der Routine, von der es aufgerufen wurde.

Ein Beispiel finden Sie bei der Erklärung, was Parameter sind.
Analog wird bei Labels und Konstanten (Anweisung CONST) verfahren. Nur Struktur-Definitionen (STRUCT-Anweisung) sind immer global.

Konzept: Parameter

In vielen Fällen muss man Werte an ein Unterprogramm übergeben, mit denen es dann arbeitet. Zum Beispiel benötigt ein Unterprogramm, das einen Namen in einer Liste suchen soll, den Namen, nach dem es suchen soll. Man könnte diesen Namen in eine globale Variable schreiben und ihn so an das Unterprogramm übergeben. Das ist besonders für Anfänger leicht zu handhaben, letztlich jedoch ein schlechter und fehleranfälliger Programmierstil.
Die bessere Lösung für dieses Problem ist, den Namen direkt an das Unterprogramm zu übergeben. Werte, die man einem Unterprogramm direkt übergeben kann, werden als "Parameter" bezeichnet.

Ein einfaches Beispiel. Wir vereinbaren eine SUB, die einen Namen mehrfach ausgeben soll:

```
SUB Namensschleife ( name$ as String, x as real)
DIM N
  For N = 1 to X
    Print name$
  NEXT N
End SUB
```

Name$ und X sind die Parameter. N ist eine lokale Variable. Die For-Schleife gibt den Namen so oft aus, wie X vorgibt. Diese Sub können wir nun beliebig oft aufrufen.

```
DIM N, T$
  T$ = "Willi"
  N = 7
  Namensschleife "Paul", 5          ' 5x Paul
  Namensschleife T$, N+3            ' 10x Willi. N ist immer noch 7
```

N und T$ seien globale Variablen. Wie oben beschrieben unterscheidet der Compiler zwischen dem globalen N und dem lokalen N in der SUB Namensschleife. Wie Sie sehen ist es völlig egal ob Sie als Parameter einen festen Wert, eine Variable oder eine Berechnung übergeben. R-BASIC wertet den Ausdruck zur Laufzeit aus und kopiert den Wert dann in die Parameter des Unterprogramms (in unserem Fall name$ und X).

Intern behandelt R-BASIC die Parameter wie lokale Variablen. Der einzige Unterschied ist, dass sie beim Aufruf des Unterprogramms mit den übergebenen Werten belegt werden. Alle anderen lokalen Variablen werden beim Aufruf des Unterprogramms gelöscht (d.h. mit Nullen belegt). Das hat wieder zwei sehr praktische Folgen:

- Sie haben die gleichen Freiheiten bei der Namensvergabe von Parametern wie bei den lokalen Variablen.
- Sie dürfen einen Parameter innerhalb eines Unterprogramms verändern ohne dass dies auf das Hauptprogramm zurückwirkt. Ändern Sie zum Beispiel unsere Sub von oben wie folgt:

```
SUB Namensschleife ( name$ as String, x as real)
DIM N
  name$ = "Der Name ist " + name$
  For N = 1 to X
     Print name$
  NEXT N
End SUB
```

und übergeben ihr dann die globale Variable T$

```
  Namensschleife T$, N+3
```

so wird die globale Variabel T$ dadurch NICHT geändert.

SUB, END SUB

Die Anweisung SUB (für Subroutine = Unterprogramm) vereinbart ein Unterprogramm. Es können Parameter an das Unterprogramm übergeben werden und innerhalb des Unterprogramms können lokale Variablen, Konstanten und Labels definiert werden, die nur innerhalb des Unterprogramms gültig (dem Compiler bekannt) sind.

Mit SUB vereinbarte Unterprogramme müssen mit END SUB (Ende der Subroutine, Leerzeichen nicht vergessen) abgeschlossen werden.
Ein vorzeitiges Verlassen des Unterprogramms mit RETURN ist möglich.

Vereinbarung: **SUB <Name>** ([**<Parameterliste>**])
<Lokale Vereinbarungen>
<**ProgrammCode>**
END SUB

<Name> Bezeichner, unter dem die SUB augerufen werden kann.
<Parameterliste> Liste Parametern, die beim Aufruf an die SUB übergeben werden sollen. Die Werte werden beim Aufruf der SUB in die Parameter kopiert.
Die Parameterliste darf leer sein. Die Klammern sind erforderlich.

Aufruf: **Name <Parameter>**

Der Aufruf gefolgt über die Angabe des Namens des Unterprogramms, gefolgt von der Parameterliste. Eine Klammer um die Parameterliste ist zulässig, aber nicht erforderlich. Die einzelnen Parameter sind durch Komma zu trennen. Existiert keine Parameterliste, wird nur der Name angegeben.

Beispiel 1:

```
SUB   Demo ()
    Print "Ich bin ein Sub-Programm"
    END SUB
```

Aufruf:

```
  Print "Im Hauptprogramm"
  Demo
  Print "Zurück im Hauptprogramm"
```

Ausgabe:

```
Im Hauptprogramm
Ich bin ein Sub-Programm
Zurück im Hauptprogramm
```

Beispiel 2:

```
SUB  PrintTableLine (x as real)    ' x ist der Parameter
DIM y, z as real                   ' y und z sind lokale Variablen
    y = x*x + 2*x – 12
    z = 12*sin(x)
    Print x, y, z
  END  SUB
```

Aufruf:

```
  For  N =  1 To 14
        PrintTableLine (N)
        Next N
```

Ausgabe:

```
 1      -9         10.098
 2      -4         10.912
 3       3          1.6934
 4       12       -9.0816
usw.
```

Beispiel 3:

```
SUB  CheckValue (a, b as real)          ' a, b: Parameter
   If  a = b Then  Return               ' Rückkehr wenn gleich
   Print "Warnung! Werte sind nicht gleich!"
 END  SUB
```

Aufruf:

```
CheckValue  X, 17     ' Warnt, wenn X nicht 17 ist
```

RETURN

Die Anweisung RETURN beendet ein Unterprogramm und kehrt zu der Routine zurück, die das Unterprogramm aufgerufen hat. RETURN in einem Actionhandler beendet die Abarbeitung des Actionhandlers und R-BASIC kehrt in den Wartezustand zurück. RETURN kann an beliebiger Stelle im Programm stehen. Insbesondere ist es erlaubt RETURN innerhalb von Schleifen und Verzweigungen zu verwenden.

Syntax:	**RETURN** Return kehrt vorzeitig aus einer SUB oder einem Actionhandler zurück.
Syntax	**RETURN <Rückgabewert>** Return mit Rückgabewert kehrt aus einer Function zurück und übergibt den Rückgabewert an die aufrufende Routine.

FUNCTION - END FUNCTION

Die Anweisung FUNCTION (für Funktion = Formel, die einen Wert berechnet) vereinbart ein Unterprogramm, das einen Wert zurückliefert. Es können Parameter an die Function übergeben werden und innerhalb der Function können lokale Variablen, Konstanten und Labels definiert werden, die nur innerhalb des Unterprogramms gültig (dem Compiler bekannt) sind.
Die Vereinbarung einer FUNCTION endet mit der Anweisung END FUNCTION (Ende der Funktion, Leerzeichen nicht vergessen).

Um eine Function zu verlassen muss die Anweisung

```
RETURN <Rückgabewert>
```

ausgeführt werden. Üblicher Weise steht diese Anweisung direkt vor der END FUNCTION Anweisung. Sie kann aber auch an beliebiger Stelle innerhalb der Function stehen. Der Rückgabewert muss dabei den in der Vereinbarung der Function angegebenen Typ haben.

Vereinbarung: **FUNCTION <Name>** (**<Parameterliste>**) **AS <Typ>**
<Lokale Vereinbarungen>
<ProgrammCode>
RETURN <Rückgabwert>
END FUNCTION

<Name> Bezeichner, unter dem die Function aufgerufen werden kann.
<Parameterliste> Liste Parametern, die beim Aufruf an die Function übergeben werden sollen. Die Werte werden beim Aufruf der Function in die Parameter kopiert.
Die Parameterliste darf leer sein. Die Klammern sind erforderlich.
<Typ> bezeichnet den Datentyp der Funktion. Es sind alle Standard-BASIC-Typen sowie selbst definierte Strukturen erlaubt. Der Rückgabewert in der RETURN-Anweisung muss diesen Typ haben.

Aufruf: **<var>** = **Name** (**<Parameter>**)

<var> ist eine Variable vom Typ, den die Funktion hat.
<Parameter> sind die Parameter - falls vorhanden - mit Komma getrennt. Die Klammern sind erforderlich, auch wenn keine Parameter existieren.

Aufruf: **Name** (**<Parameter>**)
Es ist zulässig eine Function aufzurufen, ohne den Rückgabewert zu verwenden.

Beispiel 1:
Diese einfache Funktion berechnet die Anzahl der Pixel auf dem Bildschirm. MaxX und MaxY sind globale Variablen, die die maximale x- und y-Koordinate enthalten. Da die Koordinaten bei Null beginnen müssen wir jeweils 1 addieren.

```
FUNCTION   PixelsOnScreen () As Real
  Return (MaxX+1) * (MaxY+1)
END FUNCTION
```

Aufruf:

```
DIM anz as Real
  anz = PixelsOnScreen()
```

oder

```
Print   PixelsOnScreen()
```

Beispiel 2:
Diese Funktion berechnet den Funktionswert einer linearen Funktion.

```
FUNCTION  LinFunc (x as real) As Real      ' x: Parameter
    DIM y as real                          ' y: lokale Variable
    y = 2*x + 1
    Return y
END  FUNCTION
```

Aufruf:

```
For  N = –2 To 2
    Print  N,  LinFunc(N)
    Next N
```

Ausgabe:

```
 -2          -3
 -1          -1
  0           1
  1           3
  2           5
```

Beispiel 3:
Komplexes Beispiel: Diese Funktion manipuliert eine String.

```
Function StringFunc( A$ as String, b as Real) AS String
  IF b = 0 THEN Return ""          ' Leeren String
  IF b > 0 THEN
        Return   Left$( A$, b)     ' die linken Buchstaben
  ELSE
        b = – b                    ' Aus Minus mach Plus
        Return   Right$( A$, b)    ' die rechten Buchstaben
  END IF
END FUNCTION
```

Aufruf:
Beachten Sie, dass der Compiler den Parameter A$ von der im folgenden Beispiel vereinbarten Variablen A$ unterscheidet.

```
DIM A$, B$
A$ = StringFunc("Hallo Welt", 3)
B$ = StringFunc("Hallo Welt", -3)
Print A$+B$
```

Ausgabe:

```
Halelt
```

Funktionen können nur einen einzigen Wert zurückgeben. Wenn Sie mehr als einen Wert zurückgeben wollen sollten Anfänger auf globale Variablen zurückgreifen. Fortgeschrittene Programmierer sollten eine Struktur definieren, die alle gewünschten Werte enthält und diese zurückgeben. Beispiel:

```
STRUCT Worker
  name$ As String(20)
  job$   AS String(20)
  tel    AS DWORD
END Struct


Function InitWorker() as Worker
DIM w as Worker
  w.name$ = "Pink Panther"
  w.job$ = "Spaßbolzen"
  w.tel = 47320800
  Return w
End Function
```

Actionhandler

ACTION-Handler sind Unterprogramme, die von einem Objekt direkt aufgerufen werden. Ein R-BASIC Programm besteht eigentlich aus einer Sammlung von Actionhandlern, die zu gegebener Zeit aktiviert werden. Sie können selbst wieder andere Unterprogramme (Sub, Function) aufrufen.

Innerhalb eines Actionhandlers können wie bei jedem anderen Unterprogramm lokale Variablen, Konstanten und Labels definiert werden, die nur innerhalb des Handlers gültig (dem Compiler bekannt) sind. Actionhandler müssen mit END ACTION (Ende der Aktion, Leerzeichen nicht vergessen) abgeschlossen werden.
Ein vorzeitiges Verlassen des Handlers mit RETURN ist möglich. R-BASIC geht dann wieder in den Wartezustand über.

Der Typ des Handlers beschreibt, von welchen Objekten der Handler aufgerufen werden kann.
Alle Actionhandler haben den Parameter "sender" (enthält das Objekt, dass den Handler aktiviert hat) sowie weitere, vom Typ des Handlers abhängige Parameter. Bei der Vereinbarung eines Handlers werden die Parameter NICHT explizit angegeben. Tipp: Verwenden Sie den Menüpunkt "Extras"-"Code Bausteine"-"Action-Handler". Damit erhalten Sie neben dem Handler-Rumpf einen Kommentarblock mit allen Parametern des Handlers.

Vereinbarung: <**HANDLERTYP**> **<Name>**
<Lokale Vereinbarungen>
<ProgrammCode>
END ACTION

<Handlertyp> Beschreibt, in welcher Situation und von welcher Objektklasse der Handler aufgerufen wird. Die Handlertypen sind bei den Objekten beschrieben, die sie aufrufen.
<Name> Bezeichner, der den Handler identifiziert

Beispiel

```
ButtonAction  DemoAction
  MsgBox "Button gedrückt"
END ACTION
```

Weitere Informationen zu Actionhandlern finden Sie im Objekt-Handbuch, Kapitel 1.5 (Vereinbarung von Action-Handlern) sowie bei der Beschreibung der einzelnen Objekte.

Vorab-Vereinbarung mit DECL

Sie können Unterprogramme (SUB, FUNCTION) erst dann verwenden, wenn diese zuvor dem R-BASIC-Compiler mit Namen und Parametern bekannt sind.
Damit Sie die Unterprogramme nicht in der Reihenfolge ihrer Verwendung im Quelltext anordnen müssen gibt es die DECL-Anweisung.
Die DECL-Anweisung (Declare = mache bekannt) informiert den Compiler über Namen und Parameterliste von Unterprogrammen, die erst weiter hinten im Quelltext vereinbart werden. Damit kann man

- die Übersichtlichkeit von Programmen erhöhen
- ermöglichen, dass sich Unterprogramme gegenseitig aufrufen können (A ruft B und B ruft A), was sonst nicht möglich wäre
- Libraries schreiben: Die DECL-Anweisungen gehören dann in das EXPORT-Fenster.

Syntax: **DECL SUB <Name>** (**<ParameterListe>**)
DECL FUNCTION <Name> (**<ParameterListe>**) AS **<Type>**
DECL <**HandlerType**> **<Name>**

Abarbeitung von Unterprogrammen

Die folgenden technischen Details beschreiben, wie R-BASIC intern den Aufruf von Subs und Functions organisiert. Die Kenntnis dieser Details ist für die Verwendung von Unterprogrammen nicht unbedingt erforderlich.

- Stößt R-BASIC auf eine Unterprogrammaufruf, merkt er sich die Adresse des darauffolgenden Befehls (Rücksprungadresse) und verzweigt zum Unterprogramm.
- Die Rücksprungadressen werden auf einem sogenannten Stapelspeicher (Stack) abgelegt, R-BASIC kann sich also sehr viele Adressen merken.
- Die Anweisungen RETURN bzw. END SUB beenden die Abarbeitung des Unterprogramms.
- Die zuletzt auf dem Stack abgelegte Returnadresse wird von Stack geholt und das Programm wird an dieser Stelle fortgesetzt wird. Dadurch kann Unterprogramme verschachteln, d.h. innerhalb von Unterprogrammen wieder andere Unterprogramme aufrufen. Die Verwendung eines Stacks stellt sicher, dass R-BASIC dabei immer an die korrekte Stelle zurückspringt.

Anmerkungen

- Die Anweisung End Function sollte niemals erreicht werden, weil Functions immer mit RETURN beendet werden müssen. Wird End Function erreicht handelt es sich um einen Programmierfehler und es kommt zu einem Laufzeitfehler.
- Beim Aufruf eines Actionhandlers durch ein Objekt wird ebenfalls eine (spezielle) Rücksprungadresse auf dem Stack abgelegt. Die Anweisung End Action beendet die Ausführung eines Actionhandlers. R-BASIC erkennt die spezielle Rücksprungadresse, holt sie vom Stack und kehrt in den Ruhezustand zurück.

Abwärtskompatibilität

GOSUB

R-BASIC unterstützt auch die in vielen BASIC-Dialekten verwendete Kombination GOSUB-RETURN Das kann die Übertragung fremder BASIC-Programme vereinfachen.

Die Anweisung GOSUB (Gehe zu Sub-Routine) setzt den Programmablauf an der angegebenen Stelle fort und kehrt nach Beendigung des Unterprogramms (Anweisung RETURN) wieder zurück.

Diese einfache Form der Unterprogrammtechnik hat nicht die Vorteile einer SUB oder FUNCTION (Parameterübergabe, lokale Variablen und Labels) und sollte daher in eigenen Programmen nicht verwendet werden.

Syntax: **GOSUB** <**sprungZiel**>

<sprungZiel> muss eine im Programm mit der Anweisung LABEL vereinbarte Marke oder eine im Programm explizit vergebene Zeilennummer sein.

Ein mit GOSUB aufgerufenes Unterprogramm muss mit RETURN beendet werden.

2.7 Eingaben durch den Nutzer

2.7.1 Eingabe von Text und Zahlen

In vielen Fällen muss der Nutzer während des laufenden Programms bestimmte Werte oder andere Daten eingeben. In einem objektorientierten BASIC-Programm werden dazu Textobjekte (Klassen **Memo**, **InputLine**, **VisText** oder **LargeText**) oder Objekte der Klasse **Number** (Anzeige und Eingabe von Zahlen) verwendet. Diese Objekte werden ausführlich im Objekthandbuch beschrieben.

Manchmal ist es jedoch gewünscht Daten direkt auf dem Grafikbildschirm einzugeben oder es lohnt sich nicht, wegen einer kurzen Eingabe eine Dialogbox mit einem Textobjekt zu programmieren oder ein VisText-Objekt zu verwenden. Für diesen Zweck gibt es im Standard-BASIC den Befehl **INPUT** (Eingabe direkt auf dem Grafikbildschirm) und in R-BASIC zusätzlich den Befehl **InputBox** (Eingabe in einer Dialogbox). Wenn Sie einzelne Zeichen von der Tastatur einlesen wollen, stehen Ihnen die Funktionen **InKey$**, **GetKey**, **GetKeyLP** und **GetKeyState** zur Verfügung, die im nächsten Abschnitt beschrieben werden.

INPUT

Der Befehl INPUT (= Eingabe) fordert vom Nutzer eine oder mehrere Werte an. Die Eingabe erfolgt dabei direkt auf dem Schirm. Zum Editieren stehen die Cursortasten (Pfeiltasten) links und rechts, Backspace, Pos1 und Ende zur Verfügung.

Syntax: **INPUT** [**infoString**;] **var** [, **var**] [, **var**]

infoString	(optional) Dieser Text wird ausgegeben. Es kann ein beliebiger String-Ausdruck (fester Text, Variable, Konstante, Stringfunktion) sein. Er MUSS mit einem Semikolon abgeschlossen sein. Daran erkennt R-BASIC, dass es sich um den Info-String, und nicht etwa eine einzugebende Variable handelt. Fehlt der infoString, verwendet R-BASIC ein Fragezeichen "?".
var	bezeichnet die einzugebenden Variablen. Zulässig sind alle numerischen Datentypen sowie alle String-Typen. Dazu zählen auch Feld- und Struktur-Elemente.

Beispiele:

```
INPUT A
```

```
INPUT "Bitte geben Sie Ihr Alter ein:"; A
```

```
C$ = "Was nun?"
INPUT C$; A$        ' Eingabe der Variable A$
                    ' Auf dem Schirm erscheint "Was nun?"
```

```
INPUT C$, A$          ' Eingabe der Variablen C$ UND A$
                      ' weil hinter c$ ein Koma steht.
                      ' Auf dem Schirm erscheint ein Fragezeichen.
```

Hinweise:

- INPUT ist ein Befehl um die Abwärtskompatibilität zu älteren BASIC Programmen zu gewährleisten. Sie sollen die Verwendung von Input in eigenen Programmen möglichst vermeiden.
- Es ist zu empfehlen, immer nur eine Variable anzufordern.
- Dezimaltrennzeichen ist immer der Punkt '.'
- Werden mehrere Variablen angefordert, ist als Trennzeichen das Komma ',' erforderlich. Eine Eingabe von Texten, die ein Komma enthalten, ist dann nicht möglich.
- Wird die Eingabezeile leer gelassen, so behalten die einzugebenden Variablen den Wert, den sie vorher hatten (Bestätigungsfunktion).
- Sollte als aktuelle Hintergrundfarbe BG_TRANSPARENT eingestellt sein (d.h. der Hintergrund wird bei Textausgabe nicht gelöscht) so wird während der INPUT-Anweisung ein schwarzer Hintergrund verwendet.
- Üblicher Weise ist der Screen ein BitmapContent-Objekt, wenn INPUT verwendet wird. Der Screen ist das Objekt, an das Grafik- und Textausgaben gehen (siehe Objekthandbuch, Kapitel 2.3). Input arbeitet aber auch mit anderen Objektklassen als Screen.

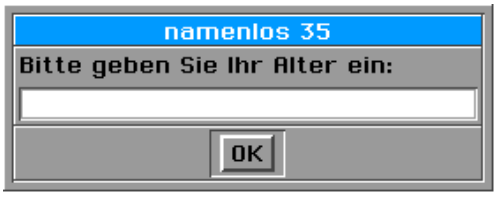
InputBox

Der Befehl InputBox (Eingabe in eine DialogBox) fordert vom Nutzer eine oder mehrere Werte an. Die Eingabe erfolgt dabei in einer Dialog-Box. Es stehen alle GEOS-typischen Editierfunktionen, einschließlich Drag & Drop (Verschieben mit der rechten Maustaste) zur Verfügung.

Syntax:	**InputBox infoString**; **var** [, **var**] [, **var**]
infoString:	Dieser Text wird ausgegeben. Es kann ein beliebiger String-Ausdruck (fester Text, Variable, Konstante, Stringfunktion) sein. Der Text wird in der Dialogbox angezeigt.
var	bezeichnet die einzugebenden Variablen. Zulässig sind alle numerischen Datentypen sowie alle String-Typen. Dazu zählen auch Feld- und Struktur-Elemente.

Beispiel:

```
InputBox   "Bitte geben Sie Ihr Alter ein:"; A
```



Hinweise: siehe INPUT

2.7.2 Direkte Abfrage der Tastatur

In vielen Fällen werden Sie zur Eingabe von Text oder Zahlen die entsprechenden R-BASIC Objekte (Klassen **Memo**, **InputLine** , **VisText**, **LargeText** oder **Number**) verwenden. Wenn Sie jedoch auf einzelne Zeichen, die über die Tastatur eingegeben werde, reagieren wollen, sollten Sie einen **Tastaturhandler** (OnKeyPressed Handler) schreiben. Tastaturhandler sind im Handbuch "Spezielle Themen", Kapitel 14 (Arbeit mit der Tastatur) beschrieben.
Für einfache Fälle und zur Wahrung der Abwärtskompatibilität stehen Ihnen zusätzlich die Funktionen **InKey$**, **GetKey**, **GetKeyLP** und **GetKeyState** zur Verfügung, die im Folgenden beschrieben werden.

InKey$

Die Funktion InKey$ (Input Keyboard = Tastatureingabe) liest ein einzelnes ASCII-Zeichen von der Tastatur ein. Bestimmte Steuerzeichen werden ebenfalls erkannt. InKey$ verwendet einen 15 Zeichen großen Puffer, damit möglichst keine Tastendrücke verloren gehen. InKey$ liefert einen Leerstring, wenn der Puffer leer ist.

Syntax: **<stringVar>** = **InKey$**

Die in der Tabelle aufgeführten symbolischen Konstanten stehen zur Verfügung, wenn die KeyCodes-Library eingebunden wird (siehe 2. Beispiel).

Erkannte Steuertasten	ASCII-Code	Symbolischer Konstante
Backspace	08 (&h08)	ASC_BS
Tabulator	09 (&h09)	ASC_TAB
Enter	13 (&h0D)	ASC_ENTER
Bild hoch	17 (&h11)	ASC_PAGE_UP
Bild runter	18 (&h12)	ASC_PAGE_DOWN
Ende	24 (&h14)	ASC_POS_END
Pos 1	25 (&h15)	ASC_POS_1
Einfg (Insert)	26 (&h16)	ASC_INS
Entf (Delete)	23 (&h17)	ASC_DEL
ESC	27 (&h1B)	ASC_ESC
Pfeiltasten (Cursortasten)		
nach unten	10 (&h0A)	ASC_DOWN
nach oben	11 (&h0B)	ASC_UP
nach links	14 (&h0E)	ASC_LEFT
nach rechts	15 (&h0F)	ASC_RIGHT

Beispiele:
Warten bis Enter gedrückt wurde

```
WHILE     InKey$  <> Chr$(13) : WEND
```

Einbinden des KeyCodes Library und warten bis Enter gedrückt wurde

```
INCLUDE "KeyCodes"
....
  WHILE  InKey$  <> Chr$(ASC_ENTER) : WEND
```

R-BASIC füllt den Tastaturpuffer bei jedem Tastendruck auf, auch wenn der Nutzer z.B. etwas in ein Textobjekt eingibt. Falls Sie nicht sicher sind, dass der Tastaturpuffer keine unerwünschten Zeichen enthält, sollten Sie den Puffer manuell leeren, beispielsweise mit der folgenden Anweisung.

```
  REPEAT  UNTIL  InKey$  = ""      ' Tastaturpuffer leeren
```

In vielen Fällen ist das allerdings nicht nötig, da der Puffer unter anderem beim Start eines jedes Handlers automatisch geleert wird.

GetKeyState

Um die aktuell gedrückten Modifier-Tasten (Shift, Ctrl, Alt) und den LockStatus (ShiftLock, NumLock, ScrollLock (=Rollen)) abzufragen, bietet R-BASIC die Funktion GetKeyState. Kenntnisse von Bit- und logischen Operationen (siehe Kapitel 2.3.4 und 2.3.5) sind für die Anwendung dieser Funktion erforderlich.

Syntax: **<numVar>** = **GetKeyState**
<numVar>: numerische Variable
Das höherwertige Byte enthält den LockStatus.
Das niederwertige Byte enthält den Shift-Status. Der Shift-Status wird auch durch die LED's an der Tastatur wiedergegeben.
Die Bits sind in der Tabelle unten erklärt.

Tabellen: Bedeutung der Bits, die von GetKeyState geliefert werden

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

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

Beispiele:

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

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

```
' Ausblenden des Lock-Staus , nur Shift-Status beachten
shiftState =  GetKeyState AND &hFF
```

Hinweis: Diese Informationen werden auch direkt an den OnKeyPressed Handler von Objekten übergeben (als Parameter keyState). Im Handbuch "Spezielle Themen", Kapitel 14, finden Sie ausführliche Informationen dazu.

GetKey

Die Funktion GetKey (= Hole Taste) fragt die Tastatur ab, ob gerade eine Taste gedrückt ist und liefert den GEOS-Tasten-Code. Dies kann ein ASCII-Code sein. Bei Tasten, denen kein ASCII-Zeichen zugeordnet ist (Steuertasten), ist es ein erweiterter Code (> 255).

GetKey liefert immer die aktuell gedrückte Taste, auch wenn diese bereits mehrfach angefragt wurde. Im Gegensatz dazu liefert InKey$ jede Taste genau einmal, sogar dann, wenn sie zum Zeitpunkt der Abfrage bereits wieder losgelassen wurde.

Syntax: **<numVar>** = **GetKey**

Beispiele:

```
' Warten bis irgendeine Taste gedrückt wurde
WHILE  GetKey = 0  : WEND
```

```
' Verzweigen, je nach Teste
' Wir verwenden mit WHILE TRUE ... WEND eine "Endlosschleife"
' die beim Drücken der Taste '3' verlassen wird
CLS
WHILE TRUE
On GetKey SWITCH
  case ASC("0"): Print "Null": End case
  case ASC("1"): Print "Eins": End case
  case ASC("2"): Print "Zwei": End case
  case ASC("3"): Print "ENDE.": BREAK    ' Schleife verlassen
  End Switch
WEND
```

Für die meisten der Steuertasten stehen symbolischen Konstanten zur Verfügung, wenn die KeyCodes-Library eingebunden wird.

Beispiel:

```
' Warten bis die Enter-Taste gedrückt
' und wieder losgelassen wurde
INCLUDE "KeyCodes"
CLS
Print "wait..."
While   GetKey <> Key_Enter  : Wend
While   GetKey = Key_Enter  : Wend
Print "Fertig."
...
```

GetKeyLP

GetKeyLP (Get Key, Last Pressed) tut weitgehend das gleiche, wie GetKey. Der einzige Unterschied ist, dass GetKeyLP, wenn es erstmalig nach dem Loslassen einer Taste gerufen wird, deren Tastencode noch liefert.

Das bedeutet konkret:

- Während eine Taste gedrückt ist, sind GetKey und GetKeyLP identisch
- Wird GetKey gerufen, nachdem die Taste losgelassen wurde, liefert es immer NULL.
- Wird GetKeyLP erstmalig gerufen, nachdem die Taste losgelassen wurde, liefert es den Tastencode der zuletzt gedrückten Taste.
- Wird GetKeyLP bei losgelassener Taste weitere Male gerufen, liefert es NULL.

Verwenden Sie GetKey, wenn Sie exakt unterscheiden wollen, ob gerade eine Taste gedrückt ist, oder nicht.

Verwenden Sie GetKeyLP oder InKey$, wenn Sie sicherstellen wollen, dass auch kurze Tastendrücke registriert werden sollen, selbst wenn Ihr Programm "beschäftigt" ist. Das können z.B. eine umfangreiche Berechnung oder der Delay-Befehl sein.

Syntax: **<numVar>** = **GetKeyLP**

Beispiel

```
DIM x
  x = GetKeyLP    ' Letzten Tastendruck löschen, falls nötig
  Print "Drücken Sie eine beliebige Taste zu Beenden!"
  Delay   60      ' Eine Sekunde Verzögerungszeit
  While GetKeyLP = 0
    Print "*"
    Delay         ' Warten bis eine Sekunde vorbei ist
    Wend
  Print "Fertig"
```

2.7.3 Messageboxen

Um auf einfache Weise Meldungen auszugeben, verfügt R-BASIC über die Funktionen **MsgBox**, **WarningBox**, **ErrorBox** und **QuestionBox**.

MsgBox

MsgBox gibt eine einfache Meldung in einer Dialogbox aus.

Syntax: **MsgBox "InfoText"**

Beispiele:

```
MsgBox "Der Prozess ist abgeschlossen"
MsgBox  A$ + " ist herausgekommen!"
```

WarningBox

WarningBox gibt eine einfache Warnung in einer Dialogbox aus.

Syntax: **WarningBox "InfoText"**

Beispiele:

```
WarningBox   "Es konnten nicht alle Daten gesichert werden."
WarningBox    A$ + " ist gefährlich"
```

ErrorBox

ErrorBox gibt eine einfache Fehlermeldung in einer Dialogbox aus.

Syntax: **ErrorBox "InfoText"**

Beispiele:

```
ErrorBox "Es ist ein Fehler aufgetreten"
ErrorBox  A$ + " ist fehlerhaft"
```

QuestionBox

QuestionBox gibt eine Frage in einer Dialogbox aus, die der User mit JA oder NEIN beantworten kann.

Syntax: **<numVar>** = **QuestionBox** (**"InfoText"**)
Die Klammern sind erforderlich.
Der Rückgabewert ist Null (R-BASIC-Konstante NO), wenn der User auf "Nein" klickt oder 1 (R-BASIC-Konstante YES), wenn der User auf "Ja" klickt

---

Beispiele:

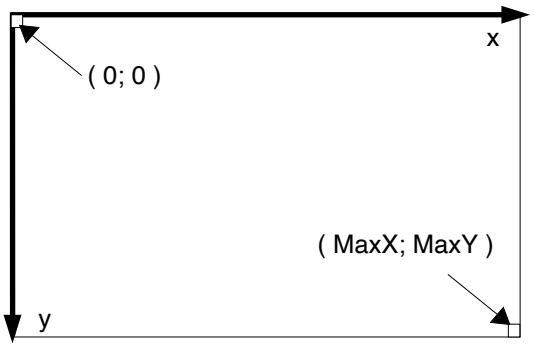
```
DIM x
  x = QuestionBox("Sind Sie sicher?")
  IF   x = NO THEN Print "Dann eben nicht"
```

```
  IF QuestionBox("Wollen Sie das Programm beenden?") = YES \
        THEN EXIT
```

2.8 Grafik

2.8.1 Das Koordinatensystem

Alle Grafikausgaben erfolgen in R-BASIC auf das aktuell eingestellte Screen-Objekt. R-BASIC verwendet das originale PC/GEOS Koordinatensystem zur Grafikausgabe. Dabei hat der Punkt links oben die Koordinaten (0; 0).



MaxX, MaxY

MaxX und MaxY sind globale Variablen, welche die größte verfügbare x- bzw. y-Koordinate des aktuellen Screen-Objekts enthalten. Wenn Sie zum Beispiel ein BitmapContent-Objekt der Größe 640x400 Pixel als aktuellen Screen haben, so gilt MaxX = 639 und MaxY = 399.

Beispiel:

```
Rectangle 0, 0, MaxX, MaxY
```

Die Werte für MaxX und MaxY vom aktuell eingestellten Screen-Objekt ab. Eventuelle Besonderheiten sind bei den entsprechenden Objekten beschrieben.

Sie haben die Möglichkeit das Koordinatensystem Ihren Wünschen anzupassen. Dazu gehört zum Beispiel, dass Sie die Lage des Koordinatenursprungs und die Skalierung der Achsen ändern können (Befehle ScreenSetTranslation und ScreenSetScale). Eine komplette Liste der Möglichkeiten finden Sie im Objekthandbuch im Kapitel 2.3.4 (Anpassen des Koordinatensystems) und für Fortgeschrittene im Kapitel 2.3.5 (Komplexe Manipulation des Koordinatensystems).

Weitere Hinweise:

- Einen kompletten Überblick über die in R-BASIC verfügbaren Möglichkeiten zur Grafikausgabe finden Sie im Objekt Handbuch, Kapitel 2.2.2 (Konzepte zur Grafikausgabe).
- Im Kapitel 2.2.1 (Objekte zur Grafikausgabe) finden Sie eine Liste aller Objekte, die als Screen arbeiten können, sowie die dazugehörigen Besonderheiten.

2.8.2 Farben

Computer beschreiben Farben durch eine Zahl. Häufig wird dabei der Rot-, der Grün- und der Blauanteil einer Farbe durch jeweils eine Zahl im Bereich von 0 (Anteil nicht vorhanden) bis 255 (Anteil mit maximaler Intensität vorhanden) beschrieben. Für diese sogenannten RGB-Farben werden 3 Byte benötigt und es ergeben sich 256*256*256 = 16.777.216 möglich Farben.
Ein anderer häufiger Fall ist, dass nur 1 Byte verwenden will, um eine Farbe zu beschreiben. Dann wird eine sogenannte Palette verwendet. Die Palette ist eine Liste von bis zu 256 Einträgen zu je drei Byte - jeweils eins für Rot, Grün und Blau. Der "Farbwert" entspricht dann der Nummer (dem sogenannten Index) des Eintrags in der Liste. Deshalb werden diese Farben auch als Index-Farben bezeichnet. Die Zählung beginnt dabei immer mit Null. Wenn keine eigene Palette vereinbart verwendet GEOS eine Standard-Palette.

R-BASIC unterstützt beide Möglichkeiten, die Verwendung von Indexfarben und die Verwendung von RGB-Farben.

Die folgende Tabelle gibt einen Überblick über die Befehle zur Farbverwaltung

Befehl	Aufgabe
Color v, h	Stellt die Vordergrund und die Hintergrundfarbe ein (1)
Ink v	Stellt nur die Vordergrundfarbe ein (1)
Paper h	Stellt nur die Hintergrundfarbe ein (1)
RGB (r, g, b)	Ermittelt den RGB-Farbewert aus den Farbanteilen
RedOf (col)	Ermittelt den Rotanteil einer RGB-Farbe
GreenOf (col)	Ermittelt den Grünanteil einer RGB-Farbe
BlueOf (col)	Ermittelt den Blauanteil einer RGB-Farbe
GrayOf (col)	Ermittelt den zu einer Farbe gehörenden Grauwert

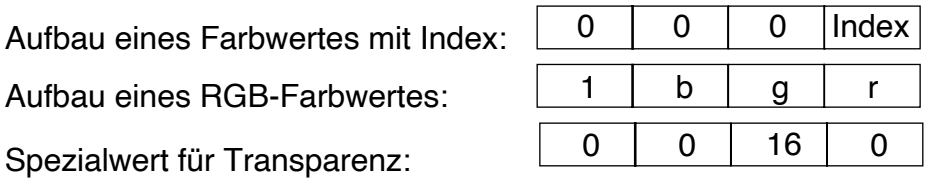
(1) Eine weitergehende Kontrolle über die verwendeten Farben, Füllmuster, Linienstile usw. haben Sie, wenn Sie die Systemvariable **graphic**, die im Kapitel 2.8.4 beschrieben ist, direkt verändern

Beschreibung von Farben in R-BASIC

Jeder Farbwert wird durch einen 32 Bit (dword) Wert beschrieben. Das kann entweder ein Farb-Index aus der GEOS-System-Palette sein (Wertebereich 0 bis 255) oder es ist ein RGB-Wert. Der numerische Wert liegt dann oberhalb von 16.777.215 (hexadezimal größer als &hFFFFFF). Diese Werte sind so gewählt, dass die Farbbefehle selbständig entscheiden können, ob es sich um eine Indexfarbe, eine RGB-Farbe oder einen Spezialfall handelt. Verwenden Sie die unten beschriebene Funktion RGB(), um einen RGB-Farbwert zu konstruieren.

Interne Details:
Zur Unterscheidung zwischen RGB- oder Index-Wert wird das höherwertige Byte des dword benutzt. Erlaubte Werte sind 0 oder 1, andere Werte können zu

unerwarteten Ergebnissen führen. Um Texte oder Blockgrafikzeichen mit transparentem Hintergrund auszugeben wird für die Hintergrundfarbe der Spezialwert BG_TRANSPARENT (=4096) eingestellt.



Farbkonstanten

Für die ersten 16 Werte der GEOS-Farbpalette existieren symbolische Namen. In vielen Fällen kann dadurch die Lesbarkeit des Programms verbessert werden. Es handelt sich einfach um die englischen Bezeichnungen der Farben.

Konstante	Wert	Farbe
BLACK	0	Schwarz
BLUE	1	Blau
GREEN	2	Grün
CYAN	3	Türkis
RED	4	Rot
VIOLET	5	Lila
BROWN	6	Braun
LIGHT_GRAY	7	Hellgrau
DARK_GRAY	8	Dunkelgrau
LIGHT_BLUE	9	Hellblau
LIGHT_GREEN	10	Hellgrün
LIGHT_CYAN	11	Helltürkis
LIGHT_RED	12	Hellrot
LIGHT_VIOLET	13	Hell-Lila
YELLOW	14	Gelb
WHITE	15	Weiß
Sonderfall		
BG_TRANSPARENT	4096	Kein Farbwert im eigentlichen Sinne. Stellt ein, dass Texte (oder Zeichen im BlockGrafik-Modus) mit transparentem Hintergrund dargestellt werden sollen. Kann nur für Hintergrundfarben verwendet werden.

Info: Die hellen Farbwerte ergeben sich, indem man zum dunkleren Farbwert 8 addiert.

COLOR

COLOR (Farbe) stellt die Farben für Vordergrund und Hintergrund ein. Die Vordergrundfarbe wird für Texte, Linien, Punkte und Flächen verwendet.

Syntax: **COLOR** **v**, **h**
v: neue Vordergrundfarbe
h: neue Hintergrundfarbe

Beispiele:

```
COLOR 7,0                    ' Hellgrau auf Schwarz
COLOR LIGHT_GRAY,BLACK       ' Hellgrau auf Schwarz
COLOR 192, 204
```

Hinweis: Um Texte oder Blockgrafik-Zeichen transparent auszugeben (d.h. der Hintergrund wird nicht gelöscht) verwenden Sie als Hintergrundfarbe den Wert BG_TRANSPARENT (=4096).

INK

INK (Tinte) stellt die Farben für den Vordergrund ein.

Syntax: **INK** **v**
v: neue Vordergrundfarbe

Beispiel:

```
INK GREEN      ' Grün, identisch mit INK 2
```

PAPER

PAPER (Papier) stellt die Farbe für den Hintergrund ein.

Syntax: **PAPER** **h**
h: neue Hintergrundfarbe

Beispiel:

```
PAPER  4   ' rot
```

Hinweis: Um Texte oder Blockgrafik-Zeichen transparent auszugeben (d.h. der Hintergrund wird nicht gelöscht) verwenden Sie als Hintergrundfarbe den Wert BG_TRANSPARENT (=4096).

RGB

Die Funktion RGB() wandelt die Farbanteile rot, grün und blau in einen R-BASIC Farbwert um.

Syntax: **<numVar>** = **RGB**(**r**, **g**, **b**)
r: Rotanteil, 0 .. 255
g: Grünanteil, 0 .. 255
b: Blauanteil, 0 .. 255

Beispiele:

```
PAPER RGB ( 50, 50, 255 )
LINE  200, 300, 180, 70, RGB ( 50, 50, 255 )
```

```
' Belegung eines Feldes der graphic-System-Variablen
graphic.textColor = RGB ( 200, 100, 100 )
```

Anmerkung:
Die Funktion RGB() verwendet folgende Formel um den Farbwert zu berechnen:
farbe = r + 256 * g + 65536 * b + 16777216
bzw. gleichwertig hexadezimal
farbe = r + &h100 * g + &h10000 * b + &h1000000

RedOf, GreenOf, BlueOf

Diese Funktionen berechnen den Rot-, Grün- bzw. Blau-Anteil eines R-BASIC Farbwertes. Der Farbwert kann ein RGB-Wert oder ein Index-Wert sein.

Syntax: **r** = **RedOf**(**farbe**)
g = **GreenOf**(**farbe**)
b = **BlueOf**(**farbe**)
farbe: Farbwert (RGB-Wert oder Index)
r, g, b: numerische Variablen, die den entsprechenden Wert aufnehmen

---

Beispiele:
Die Systemvariable graphic ist im Kapitel 2.8.4 beschrieben.

```
r = RedOf( graphic.lineColor )
g = GreenOf( graphic.areaColor )
b = BlueOf( C_YELLOW )
```

GrayOf

Diese Funktion berechnet den Grauwert (d.h. die Helligkeit) zu einer Farbe. Das unterschiedliche Helligkeitsempfinden des Auges für den Rot-, Grün- und Blau-Anteil wird berücksichtigt. Der Farbwert kann ein RGB-Wert oder ein Index-Wert sein.

Syntax: **<numVar>** = **GrayOf**(**farbe**)
farbe: Farbwert (RGB-Wert oder Index)

Beispiele:

```
h = GrayOf( graphic.textColor )
h = GrayOf( PGet ( x, y) )
h = GrayOf( C_YELLOW )
```

2.8.3 Linien, Punkte und Figuren

Die folgende Tabelle gibt eine Übersicht über die in R-BASIC verfügbaren Grafikbefehle zu Linien, Punkten und Figuren.

Befehl / Strukturtyp	Aufgabe
CLS	Löscht den Bildschirm
LINE x, y, xe, ye [, f]	Zeichnet eine gerade Linie
PSet x, y [, f]	Zeichnet einen Punkt in der Vordergrundfarbe
PReset x, y	Zeichnet einen Punkt in der Hintergrundfarbe
PGet (x, y)	Liest die Farbe eines Punktes einer Bitmap aus
Circle x, y, r [, f]	Zeichnet einen ungefüllten Kreis
Ellipse x, y, xe, ye [, f]	Zeichnet eine Ellipse
FillEllipse x, y, xe, ye [, f]	Zeichnet eine gefüllte Ellipse
Rectangle x, y, xe, ye [, f]	Zeichnet ein Rechteck
FillRect x, y, xe, ye [, f]	Zeichnet ein gefülltes Rechteck
PointList	Strukturtyp für Polyline, Polygon und Splines
PolyLine pl	Zeichnet einen verbundenen Linienzug
Polygon pl	Zeichnet einen geschlossenen Linienzug
FillPolygon pl	Zeichnet ein gefülltes Polygon
Spline pl	Zeichnet eine Kurve durch mehrere Punkte
ClosedSpline pl	Zeichnet eine geschlossene Kurve
BezierSpline pl	Für Fortgeschrittene: zeichnet eine Kurve

CLS

CLS (Clear Screen) löscht den Bildschirm mit der aktuellen Hintergrundfarbe.

Syntax: **CLS**

LINE

Der Befehl LINE (Linie) zeichnet eine Line auf dem Schirm. Standardmäßig wird die aktuelle Vordergrundfarbe verwendet. Wird der Parameter f angegeben, so wird diese Farbe verwendet.

Syntax: **LINE x0, y0, x1, y1** [, **f**]
x0, y0: Startpunkt der Linie
x1, y1: Endpunkt der Linie
f: Linienfarbe (optional, Indexfarbe oder RGB-Farbe)

Beispiel:

```
LINE 0, 0, 100, 200, 15 ' zeichnet eine weiße Linie
```

PSet

Der Befehl PSet (Point Set = Punkt Setzen) setzt einen Punkt auf dem Schirm. Standardmäßig wird die aktuelle Vordergrundfarbe verwendet. Wird der Parameter f angegeben, so wird diese Farbe verwendet.

Syntax: **PSet** **x**, **y** [, **f**]
x, y: Koordinaten des Punktes
f: Punktfarbe (optional, Indexfarbe oder RGB-Farbe)

Beispiel:

```
PSet 19, 200, 12    ' setzt einen roten Punkt
```

PReset

Der Befehl PReset (Point Reset = Punkt Zurücksetzen) löscht einen Punkt auf dem Schirm, d.h. der Punkt wird mit der Hintergrundfarbe belegt.

Syntax: **PReset** **x**, **y**
x, y: Koordinaten des Punktes

Beispiel:

```
PReset 19, 200
```

PGet

Die Funktion PGet (Point Get = Punkt holen) liefert den Farbcode des Bildpunktes an den gegebenen Koordinaten. PGet ist ein Kompatibilitätsbefehl, er setzt voraus, dass der aktuelle Screen ein BitmapContent Objekt ist.

Syntax: **<numVar>** = **PGet** (**x**, **y**)
x, y: Koordinaten des Punktes

Beispiel:

```
DIM f
  f = PGet ( 20, 100 )
```

CIRCLE

Der Befehl CIRCLE (Kreis) zeichnet einen ungefüllten Kreis auf dem Schirm. Für einen gefüllten Kreis verwenden Sie den Befehl FillEllipse.

Syntax: CIRCLE **x0**, **y0**, **r** [, **f**]
x0, y0: Koordinaten des Mittelpunkts
r: Radius des Kreises
f: Linienfarbe (optional, Indexfarbe oder RGB-Farbe)

Beispiel:

```
CIRCLE 100, 200, 50
```

ELLIPSE

Der Befehl ELLIPSE zeichnet eine ungefüllte Ellipse auf dem Schirm.

Syntax: **ELLIPSE** **x0**, **y0**, **x1**, **y1** [, **f**]
Die Koordinaten beschreiben das einschließende Rechteck
x0, y0: eine Ecke (z.B. links unten oder links oben)
x1, y1: gegenüberliegende Ecke
f: Linienfarbe (optional, Indexfarbe oder RGB-Farbe)

Beispiel:

```
ELLIPSE 0, 0, 200, 50, BLUE     'eine blaue, längliche Ellipse
```

RECTANGLE

Der Befehl RECTANGLE (Rechteck) zeichnet ein ungefülltes Rechteck auf dem Schirm.

Syntax: RECTANGLE **x0**, **y0**, **x1**, **y1** [, **f**]
x0, y0: eine Ecke (z.B. links unten)
x1, y1: gegenüberliegende Ecke
f: Linienfarbe (optional, Indexfarbe oder RGB-Farbe)

Beispiel:

```
RECTANGLE  100, 100, 200, 200, WHITE     ' ein weißes Quadrat
```

FillEllipse

Der Befehl FillEllipse (Fülle Ellipse) zeichnet eine gefüllte Ellipse auf dem Schirm.

Syntax: FillEllipse **x0**, **y0**, **x1**, **y1** [, **f**]
Die Koordinaten beschreiben das einschließende Rechteck
x0, y0: eine Ecke (z.B. links unten)
x1, y1: gegenüberliegende Ecke
f: Flächenfarbe (optional, Indexfarbe oder RGB-Farbe)

Beispiel:

```
ELLIPSE 0, 0, 200, 50, BLACK    ' eine schwarze Ellipse
```

FillRect

Der Befehl FillRect (Fülle Rechteck) zeichnet ein gefülltes Rechteck auf dem Schirm.

Syntax: FillRect **x0**, **y0**, **x1**, **y1** [, **f**]
x0, y0: eine Ecke (z.B. links unten)
x1, y1: gegenüberliegende Ecke
f: Flächenfarbe (optional, Indexfarbe oder RGB-Farbe)

Beispiel:

```
FillRect  100, 100, 200, 200, YELLOW    ' ein gelbes Quadrat
```

PointList

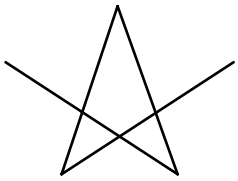
Die Struktur PointList enthält eine Liste von Punkten um Polygone, verbundene Linien und Splines zu zeichnen.

```
STRUCT PointList
  numPoints          as INTEGER
  xOffset, yOffset   as INTEGER
  x(31)              as INTEGER
  y(31)              as INTEGER
  End Struct
```

Feld	Bedeutung, gültige Werte
numPoints	Anzahl der gültigen Koordinatenpaare in der Liste Erlaubte Werte: 2 ... 32
xOffset, yOffset	Zusätzliches Offset für die Zeichenposition der Figur. Diese Werte werden zu jedem Koordinatenpaar addiert, bevor die Figur gezeichnet wird.
x(31), y(31)	Koordinatenpaare von bis zu 32 Punkten, aus denen die Figur gebildet wird. (x(0)/y(0) bis x(31)/y(31))

PolyLine

PolyLine zeichnet einen offenen Linienzug aus mehreren Geraden in der aktuellen Vordergrundfarbe.



Syntax: **PolyLine <pl>**
<pl>: Variable oder Ausdruck vom Typ PointList

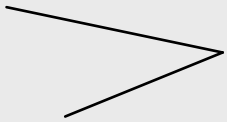
Erläuterungen zur Struktur <pl>:

- Die Elemente x(0), y(0) bis x(31), y(31) enthalten die Koordinatenpaare der Ecken des Linienzugs.
- numPoints enthält die Anzahl der gültigen Koordinatenpaare. Es werden also numPoints - 1 Linien gezeichnet.
- Die Elemente xOffset und yOffset ermöglichen es, die ganze Figur an eine andere Stelle zu zeichnen, ohne alle Koordinaten einzeln ändern zu müssen. Dazu werden die Werte von xOffset und yOffset vor jeder Zeichenoperation zu den einzelnen Koordinaten addiert.
  "Polyline p" entspricht also der folgenden BASIC-Sequenz

```
 LINE  p.x(0)+xOffset  , p.y(0)+yOffset , p.x(1)+xOffset ,
 p.y(1)+yOffset
 LINE  p.x(1)+xOffset  , p.y(1)+yOffset , p.x(2)+xOffset  ,
 p.y(2)+yOffset
.. usw
```

Das folgende Beispiel zeichnet eine PolyLine mit 3 Punkten.

```
Dim p as PointList
  p.x(0) = 10 : p.y(0) = 10
  p.x(1) = 100 : p.y(1) = 20
  p.x(0) = 30 : p.y(0) = 50
  p.numPoints = 3
  PolyLine p
```



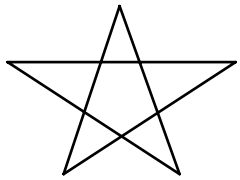
Dieser sehr einfache Fall entspricht den folgenden BASIC Befehlen:

```
 LINE 10, 10, 100, 20
 LINE 100, 20, 30, 50
```

Ein weiteres Beispiel finden Sie in der Datei: R-BASIC\Beispiele\Grafik\Polygon Demo

Polygon

Polygon zeichnet einen geschlossenen Linienzug aus mehreren Geraden in der aktuellen Vordergrundfarbe. Dazu wird der letzte Punkt der PointList mit dem ersten Punkt der PointList verbunden.

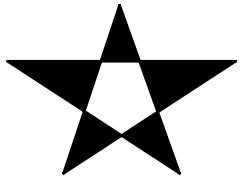


Syntax: **Polygon <pl>**
<pl>: Variable oder Ausdruck vom Typ PointList

Erläuterungen zur Struktur <pl>: Siehe PolyLine
Für ein Beispiel siehe Datei: R-BASIC\Beispiele\Grafik\Polygon Demo

FillPolygon

FillPolygon zeichnet einen gefüllten, geschlossenen Linienzug aus mehreren Geraden in der aktuellen Vordergrundfarbe.

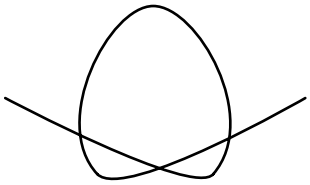


Syntax: **FillPolygon <pl>**
<pl>: Variable oder Ausdruck vom Typ PointList

Erläuterungen zur Struktur <pl>: Siehe PolyLine
Für ein Beispiel siehe Datei: R-BASIC\Beispiele\Grafik\Polygon Demo

Spline

Spline zeichnet einen glatten Linienzug durch die gegebenen Punkte in der aktuellen Vordergrundfarbe.

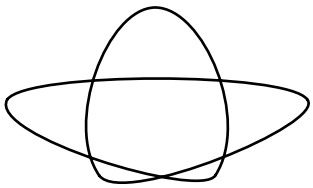


Syntax: **Spline <pl>**
<pl>: Variable oder Ausdruck vom Typ PointList

Erläuterungen zur Struktur <pl>: Siehe PolyLine
Für ein Beispiel siehe Datei: R-BASIC\Beispiele\Grafik\Polygon Demo

ClosedSpline

ClosedSpline zeichnet einen geschlossenen glatten Linienzug durch die gegebenen Punkte in der aktuellen Vordergrundfarbe.



Syntax: **ClosedSpline <pl>**
<pl>: Variable oder Ausdruck vom Typ PointList

Erläuterungen zur Struktur <pl>: Siehe PolyLine
Für ein Beispiel siehe Datei: R-BASIC\Beispiele\Grafik\Polygon Demo

BezierSpline

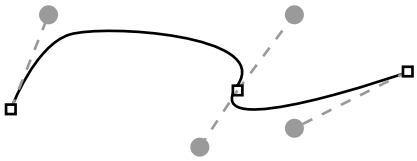
Achtung! BezierSpline ist eine Anweisung für fortgeschrittene Programmierer!

Syntax: **BezierSpline <pl>**
<pl>: Variable oder Ausdruck vom Typ PointList

BezierSpline zeichnet einen Linienzug durch die gegebenen Punkte in der aktuellen Vordergrundfarbe, wobei die einzelnen Segmente durch ihren Anfangs- und Endpunkt sowie durch 2 Kontrollpunkte beschrieben werden. In den folgenden Bildern sind die Kurvenpunkte durch Vierecke, die Kontrollpunkte durch graue Kreise markiert. Beide dienen der Illustration, sie werden nicht mit gezeichnet.

Linkes Bild: Ein einzelnes Kurvensegment, bestehend aus 2 Kurvenpunkten und 2 Kontrollpunkten. Für diese Figur müssen 4 Punkte an BezierSpline übergeben werden.

Rechtes Bild: Zwei Kurvensegmente und die dazugehörigen Kontrollpunkte. Für diese Figur müssen 7 Punkte an BezierSpline übergeben werden.



Die an BezierSpline übergebende PointList Struktur muss die Koordinaten der Punkte in folgender Reihenfolge enthalten:
Kurve, control, control, Kurve, control, control, Kurve, control, control, ... Kurve.
Das Feld numPoints enthält die Gesamtzahl der übergebenen Punkte. Diese Zahl muss der Beziehung 3*n+1 entsprechen, wobei n die Anzahl der Kurvensegmente ist. Die Anzahl der Punkte auf der Kurve ist damit n+1.

Hinweise:

- BezierSpline macht genau das Gleiche wie das Spline-Werkzeug  in GeoDraw.
- Die Anweisungen Spline und ClosedSpline nutzen intern die gleiche Funktion wie BezierSpline. Sie berechnen sich ihre Kontrollpunkte jedoch selbst.
- Sie können mit BezierSpline maximal 10 Kurvensegmente auf einmal zeichnen. Das entspricht 31 zu übergebenden Punkten.
- Um eine Ecke an einem Punkt zu erzeugen können Sie die Koordinaten der beiden Kontrollpunkte links und rechts von diesem Punkt auf die gleichen Koordinaten wie die des Eckpunktes setzen.
- Ein einzelnes Kurvensegment nennt man Bézierkurve. GEOS benutzt kubische Bézierkurven. Weitere Informationen zu Bézierkurven finden Sie im Internet.
- Weitere Erläuterungen zur Struktur <pl>: Siehe PolyLine

2.8.4 Die Systemvariable "graphic": Mixmodes und mehr

Alle Parameter des Grafik-Systems vom R-BASIC lassen sich über die Felder der Systemvariablen **graphic** einstellen. Die Verwendung dieser Variablen gestattet einen wesentlich detaillierten Zugriff auf die Grafik-Eigenschaften, als die Befehle COLOR, PAPER und INK. GEOS verwaltet getrennte Farben für Text, Linien und Flächen. R-BASIC speichert zusätzlich noch eine Farbe für den Hintergrund (z.B. von Textausgaben, siehe PAPER-Befehl).
graphic ist vom Typ **GraphicDrawStruct**, der folgendermaßen definiert ist:

```
STRUCT  GraphicDrawStruct
    mixMode               AS Word
    backColor             AS DWord
    lineColor             AS DWord
    lineDrawMask          AS Word
    lineWidth, lineStyle  AS Word
    lineEnd, lineJoin     AS Word
    areaColor             AS DWord
    areaDrawMask          AS Word
    textColor             AS DWord
    textDrawMask          AS Word
    drawFlags             AS Word
    reserve(6)            AS Word  ' reserviert für zukünftige
                                   ' Erweiterungen
END STRCUT
```

Die folgende Tabelle enthält die Bedeutung der einzelnen Felder, sortiert nach der Wichtigkeit / Häufigkeit ihrer Verwendung.

Häufig verwendete Felder

Feld	Bedeutung
areaColor lineColor textColor	Farbe für Flächen (areaColor), Linien (lineColor) und Texte (textColor). Die Befehle COLOR und INK setzen alle drei Farben auf den gleichen Wert.
backColor	Hintergrund-Farbe für Texte. Wird von den Befehlen PAPER und COLOR belegt. Um Texte oder Blockgrafikzeichen mit transparentem Hintergrund auszugeben wird der Spezialwert BG_TRANSPARENT (=4096) verwendet.
mixMode	Schreibmodus für Flächen und Linien. Wirkt nicht auf Texte. Beschreibt, auf welche Weise neue Grafikausgaben mit bereits vorhandenen verknüpft werden. Standard MM_COPY, häufig verwendet: MM_XOR und MM_INVERT.
lineWidth	Liniendicke
lineStyle	Linienstil, z.B. gestrichelt. In der Tabelle unten finden Sie die zulässigen Werte und ihre Bedeutung.

Weniger häufig verwendete Felder

Feld	Bedeutung
areaDrawMask	Füllmuster für Flächen. Die Fläche wird mit einen Muster hinterlegt, das von GEOS erzeugt wird. R-BASIC definiert einige Konstanten zur Arbeit mit Füllmustern. In der Tabelle unten finden Sie die zulässigen Wertebereiche und ihre Bedeutung sowie ein paar Beispiele.
lineDrawMask	Füllmuster für Linien. Details siehe areaDrawMask.
textDrawMask	Füllmuster für Text. Details siehe areaDrawMask. Selten verwendet, da Texte i.a. sehr klein sind.
lineEnd	Linienabschluss. Erlaubte Werte: 0: Normales Ende, 1: Halbrund, 2: Quadrat Die Werte 1 und 2 verlängern die Linie etwas.
lineJoin	Verbindung zwischen Linien bei einer Figur (Rechteck). erlaubte Werte: 0: Normal (eckig), 1: abgerundet, 2: abgeflacht.
drawFlags	Diverse Flags. Aktuell verfügbar: GDF_SCALE_PSET: Bewirkt, dass die von PSet und PReset gesetzten Punkte als Flächen gezeichnet werden. Damit werden sie Punkte vergrößert, wenn der Screen skaliert ist.

Hinweis: Während der Ausführung einer PRINT-Anweisung werden einige Felder der graphic-Variablen intern zeitweise geändert, dann aber wieder zurückgesetzt. Das kann bedeutsam sein, wenn die Printliste Funktionsaufrufe enthält.

Tabelle der Linienstile: Erlaubte Werte für das Feld lineStyle

Wert	Konstante	Bedeutung
0	LS_SOLID	durchgehend
1	LS_DASHED	gestrichelt
2	LS_DOTTED	gepunktet
3	LS_DASHDOT	Strich-Punkt
4	LS_DASDDOT	Strich-Doppelpunkt

Tabelle der Füllmuster: Erlaubte Werte für die Felder areaDrawMask, lineDraw-Mask und textDrawMask. Beispiele für Füllmuster finden Sie in "R-BASIC Anhänge", Abschnitt C.

Wert	Konstante	Bedeutung
0 - 24	–	Von GEOS bereitgestellte Muster, siehe unten.
25	DM_100	"Normalzustand", 100% Deckung.
26 - 88	–	Unterschiedliche "Transparenzgrade". Größere Werte entsprechen höherer Transparenz.
89	DM_0	Null % Deckung, vollständig transparent.
128	DM_INVERSE	Wird zu einem der anderen Werte addiert. Das Muster wird invertiert.

Der Mix-Mode

Im Normalfall geht man davon aus, dass neu gezeichnete Linien oder Flächen vorhandene Grafiken überschreiben. Das muss aber nicht so sein. Mit dem Feld graphic.mixMode können Sie bestimmen, wie neu gezeichnete Linien oder Flächen mit dem bereits vorhandenen Hintergrund verknüpft werden sollen. Dabei wird der Farbwert jedes Pixels ermittelt, indem der Farbwert des an dieser Stelle bereits vorhandenen Pixels über eine logische Operation mit dem Farbwert der Zeichenfarbe verknüpft wird.

Von besonderer Bedeutung sind die Modi MM_XOR und MM_INVERT. In diesen Modi wird z.B. eine Linie beim ersten Zeichnen erscheinen und beim nochmaligen Zeichnen wieder gelöscht. So kann man z.B. "Gummi-Linien" realisieren, ohne den Hintergrund zu beschädigen. Im Zweifelsfall verwenden Sie MM_INVERT.

Achtung! graphic.mixMode wirkt nicht bei Textausgaben (PRINT) und bei der Ausgabe von Bildern (DrawImage, DrawPicture, DrawBitmap, DrawIcon).

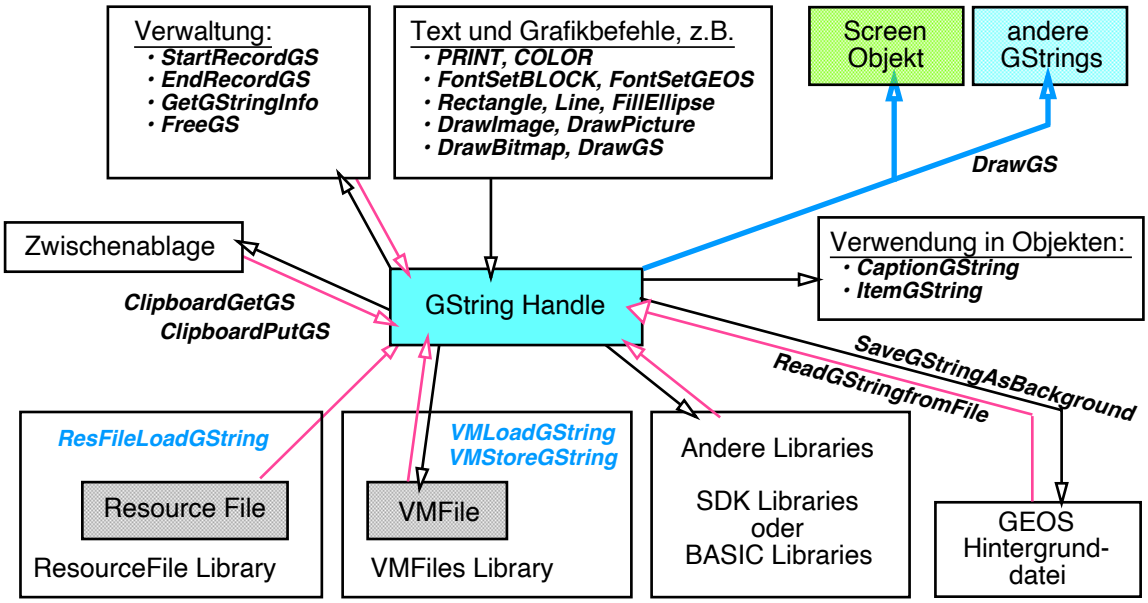
Tabelle der Mix-Modes: Erlaubte Werte für das Feld mixMode

Wert	Konstante	Bedeutung
0	MM_CLEAR	Das Zeichnen einer Grafik löscht den überschriebenen Bereich. Die Zeichenfarbe spielt keine Rolle.
1	**MM_COPY**	**Standardwert. Die neue Grafik überschreibt vorhandene Grafiken.**
2	MM_NOP	Die Grafikausgabe wird ignoriert.
3	MM_AND	Die Farben in dem überschriebenen Bereich werden logisch AND mit der Zeichenfarbe verknüpft.
4	MM_ INVERT	Die Farben in dem überschriebenen Bereich werden logisch invertiert. Die Zeichenfarbe spielt keine Rolle. Dieser Modus wird häufig benutzt.
5	MM_XOR	Die Farben in dem überschriebenen Bereich werden logisch XOR mit der Zeichenfarbe verknüpft. Dieser Modus wird häufig benutzt.
6	MM_SET	Das Zeichnen einer Grafik setzt den überschriebenen Bereich auf schwarz. Die Zeichenfarbe spielt keine Rolle.
7	MM_OR	Die Farben in dem überschriebenen Bereich werden logisch OR mit der Zeichenfarbe verknüpft.

Der Mix-Mode verwendet logische Operationen (AND, OR, XOR) zur Verknüpfung der Farbwerte. Für System-Farben (beschrieben durch einen Index) wirkt die Verknüpfung auf den Index. Deshalb hängen die Ergebnisse einiger Mix-Modi davon ab, welche Farbtiefe verwendet wird. Das kann insbesondere in dem häufigen Fall, dass eine 256-Farb-Bitmap gemeinsam mit einem True-Color Bildschirm verwendet wird, zu unerwarteten Ergebnissen führen (die Resultate in der Bitmap und auf dem Bildschirm unterscheiden sich). Hier hilft nur ausprobieren, ob das Ergebnis für die eigenen Zwecke geeignet ist oder nicht. R-BASIC reicht an dieser Stelle einfach das vom GEOS-System bereitgestellte Verhalten durch.

2.8.5 Arbeit mit Graphic Strings

Ein Graphic String (im Folgenden kurz GString) ist eine Folge von Grafikbefehlen oder Textausgaben, die gemeinsam gespeichert werden. Dieser GString kann später beliebig oft "abgespielt" werden. Dabei werden die enthaltenen grafischen Kommandos mit hoher Geschwindigkeit ausgeführt, viel schneller als dies als Folge von BASIC-Anweisungen möglich ist. Das folgende Bild gibt einen Überblick über die Möglichkeiten, die R-BASIC zur Arbeit mit GStrings bietet.



Es gibt mehrere Möglichkeiten an eine GString zu kommen. In vielen Fällen werden Sie ihn selbst aufzeichnen. Dazu müssen Sie zunächst mit **StartRecordGS** die Aufzeichnung starten. StartRecordGS liefert ein Handle zurück, mit dem Sie später den GString wiedergeben können. Intern wird der GString in einer Datei gespeichert, die R-BASIC zur Verfügung stellt. Deswegen müssen Sie die ungefähre Datenmenge, die der GString aufnehmen soll, angeben.

Ab diesem Zeitpunkt gehen alle Grafik- und Textausgaben, die sonst auf den Bildschirm gehen würden, in den GString und werden aufgezeichnet. Es sind grundsätzlich alle Text- und Grafikbefehle erlaubt. Das schließt explizit die Wiedergabe anderer GStrings ein. Die Grafikbefehle werden dabei in den neuen GString kopiert. Mit **EndRecordGS** wird der Aufzeichnungsmodus beendet.

Der GString kann nun mit **DrawGS** ausgegeben werden. Die Routine **GetGStringInfo** liefert Informationen über einen GString, z.B. seine Abmessungen. Damit können Sie ihn z.B. zentriert oder rechtsbündig an eine bestimmte Position zeichnen. Außerdem können Sie GStrings mit **CaptionGString** als grafische Captions für Objekte und mit **ItemGString** ( ) als grafische Listeneinträge in DynamicList Objekten verwenden.

Statt einen GString selbst aufzuzeichnen können Sie ihn mit **ClipboardGetGS** aus der Zwischenablage zu holen oder mit **ReadGStringFromFile** aus einer GEOS Hintergrunddatei lesen. Außerdem bieten die R-BASIC Libraries "VMFiles" und "ResFile" Funktionen an, einen GString in eine Datei zu schreiben bzw. ihn von

dort zu laden. Sie haben weiterhin die Möglichkeit einen GString mit **ClipboardPutGS** in die Zwischenablage zu kopieren.

Wenn Sie den GString nicht mehr benötigen **müssen** Sie ihn **meist** mit **FreeGS** freigeben. Beachten Sie dazu die Dokumentation der Routine, die den GString angelegt hat! Mit FreeGS wird der GString aus der Datei, die ihn enthält, gelöscht. Beachten Sie, dass ein GString bei einem System Shutdown nicht automatisch gelöscht wird, das Handle auf ihn steht nach einem Systemneustart aber nicht mehr zur Verfügung. Sie sollten deswegen alle "globale" GStrings, die von einer Routine angelegt und von anderen verwendet werden, im OnExit-Handler des Application-Objekts freigeben.

StartRecordGS

StartRecordGS beginnt die Aufzeichnung eines GStrings. Der GString wird als unsichtbarer Screen gesetzt, die aktuellen Screendaten werden gesichert und nach EndRecordGS wieder hergestellt. Alle Text- und Grafikausgaben gehen ab sofort in den GString und werden aufgezeichnet.
Dabei gelten anfangs die folgenden Einstellungen:

- Textfont: Es wird der Standard-Font eingestellt: Fontmode Fixed, FID_MONO, 14 Punkt.
- Farben: Die Vordergrundfarbe wird auf Schwarz, die Hintergrundfarbe wird auf Transparent gestellt. Texte werden also transparent ausgegeben.
- Änderung des Fonts, der Textgröße, der Farben und anderer Grafikeigenschaften während der Aufzeichnung des GStrings haben keine Auswirkungen auf andere Teile des Programms.
- Die Blockfonts sind immer global. Das heißt, dass geladene Fonts nach dem Aufruf von FontSetBlock zur Verfügung stehen und dass diesbezügliche Änderungen auch nach dem Aufruf von EndRecordGS bestehen bleiben und so andere Teile des Programms beeinflussen können.
- GStrings haben prinzipiell keine Begrenzung. Die globale Variablen MaxX und MaxY sind ohne Bedeutung.

Syntax: **<hanVar>** = **StartRecordGS** (dataSize)
<hanVar> = **StartRecordGS** ( ) ' entspricht DS_TINY
' die Klammern sind erforderlich!

<hanVar> Variable vom Typ HANDLE
Speichert die Referenz auf den GString

Das von StartRecordGS zurückgegeben Handle wird für die anderen GString-Befehle benötigt. Der Parameter dataSize bestimmt die ungefähre Größe der GStringdaten. Damit kann R-BASIC abschätzen, wieviel Platz es in der Datei, die den GString aufnehmen soll, reservieren muss. Wenn sich schon viele Daten in der Datei befinden (z.B. Bitmaps von BitmapContent-Objekten oder andere GStrings) kann R-BASIC gegebenen Falls eine neue Datei anlegen. Allerdings ist der Wert nicht kritisch. Geben Sie DS_TINY an und verbrauchen trotzdem ein Megabyte passiert im Allgemeinen nichts. Haben Sie aber viele GStrings, Bitmap-

Objekte oder Objekte im gepufferten Modus (z.B. Canvas mit buffered = TRUE gesetzt) gleichzeitig sollten Sie dem Wert etwas mehr Aufmerksamkeit widmen.

Der Defaultwert für dataSize ist DS_TINY. Die folgende Tabelle enthält die zulässigen Werte.

Konstante	Wert	Zu erwartende Datenmenge
DS_TINY	0	nicht mehr als 10 .. 20 kByte
DS_SMALL	1	nicht mehr als 50 .. 100 kByte
DS_MEDIUM	2	nicht mehr als 500 kByte ... 1 MB
DS_LARGE	3	nicht mehr als 5 MByte
DS_HUGE	4	möglicherweise mehr als 5 MByte

Beispiele:
1. Normale Grafikbefehle wie Line, Rectangle, FillEllipse usw. erfordern jeweils 10 bis 15 Byte. Texte erfordern pro Zeichen 1 Byte. Für die meisten Fälle ist daher der Defaultwert DS_TINY völlig ausreichend. Das entspricht ca. 1000 Zeichenbefehlen.
2. Block-Font Grafiken erfordern 1 Byte pro Pixel (256-Color Grafiken) oder nur 1 Byte auf 8 Pixel (monochrome Grafiken). Ein GString mit 100 Blockfont-Grafiken der Größe 32x32 Pixel erfordert, wenn es sich um 256-Color Grafiken handelt, 100x32x32 = 102400 Byte (100 kByte). Verwenden Sie DS_SMALL oder DS_MEDIUM.
3. Eine Bitmap der Farbtiefe 8 Bit erfordert 1 Byte pro Pixel. Für eine Bitmap der Größe 640x480 Pixel (=307200 Byte) verwenden Sie DS_MEDIUM.
4. True-Color-Bitmaps benötigen 3 Byte pro Pixel. Für eine 800x600 True-Color-Bitmap ist DS_LARGE angebracht.

EndRecordGS

EndRecordGS beendet die Aufzeichnung eines GString. Die vor dem Aufruf von StartRecordGS geltenden Screen- und Grafikeinstellungen werden wieder hergestellt. Ab sofort kann der GString verwendet werden.

Syntax: **EndRecordGS** **<han>**
<han> Handle, das von StartRecordGS geliefert wurde.

DrawGS

DrawGS zeichnet den GString an die Position x, y.

Syntax: **DrawGS** **<han>** , **x** , **y**
<han>: Handle, das von StartRecordGS geliefert wurde.
x, y: Zeichenposition in Pixeln. Die linke obere Ecke des GString wird an diese Position gezeichnet.

FreeGS

FreeGS gibt das Handle und den GString wieder frei. Der vom GString belegte Speicherplatz wird freigegeben.

Syntax: **FreeGS** **<han>**
<han> Handle, das von StartRecordGS, ClipboardGetGS oder ReadGStringFromFile geliefert wurde.

Tipp: Um sicherzustellen, dass das System nicht crasht, falls das Handle irrtümlich noch einmal mit DrawGS verwendet wird können Sie es nach dem Freigeben durch FreeGS () mit der Anweisung "han = NullHandle()" löschen. R-BASIC gibt dann bei einer irrtümlichen Verwendung mit DrawGS nur eine entsprechende Meldung aus.

Wichtig: Wenn Sie eine Library-Routine verwenden um einen GString zu erzeugen, lesen Sie bitte die Dokumentation der Routine sorgfältig, um zu entschieden, ob Sie diesen GString mit FreeGS freigeben müssen oder nicht!

Beispiel: Die Routine verwendet einen GString, um einem Objekt eine Grafik als Caption zuzuweisen. Dazu verwenden wir die Instancevariable CaptionGString. Das ist ein üblicher Weg um einfache Grafiken, die zur Laufzeit gelegentlich geändert werden müssen, darzustellen. Die Zeile "MyObj.CaptionGString = gsHan" kopiert den GString in das Objekt, so dass wir ihn mit "FreeGS gsHan" wieder freigeben können (und müssen!).

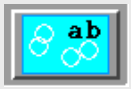
```
SUB SetCaption ( )
DIM gsHan as HANDLE

  gsHan = StartRecordGS ( )
  FillRect 0, 0, 48, 32, LIGHT_BLUE
  Rectangle 0, 0, 48, 32, BLUE
  INK WHITE
  Ellipse 4, 12, 14, 22
  Ellipse 7, 5, 17, 15
  Ellipse 22, 18, 32, 28
  Ellipse 32, 14, 42, 24
  Ink BLACK
  printfont.style = TS_BOLD
  Print atxy 25,1;"ab"
  EndRecordGS gsHan

  MyObj.CaptionGString = gsHan

  FreeGS gsHan

End SUB
```



Beispiel: Verwendung eines GString, dessen Handle in einer globalen Variable gespeichert ist. Der GString wird beim Schließen des Programms freigegeben.

Definition der globalen Variablen

```
DIM globalGS AS HANDLE
```

UI-Code Ausschnitt

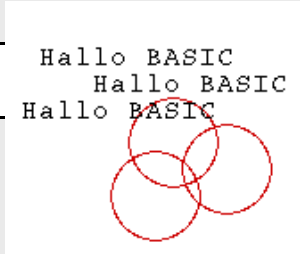
```
Application DemoApplication
  <.....>
  OnExit = ExitHandler
End Object
```

Anlegen des GString

```
SUB CreateGS ( )
  globalGS = StartRecordGS()
  Ellipse 50, 50, 100, 100, RED
  Print atxy 0, 20; "Hallo BASIC"
  EndRecordGS ( globalGS )
END SUB
```

Verwendung des GString

```
SUB DrawGlobalGS()
  DrawGS globalGS, 20, 30
  DrawGS globalGS, 10, 60
  DrawGS globalGS, 50, 45
End SUB
```



Zugehöriger OnExit-Handler zum freigeben des GString.
FreeGS ignoriert leere Handles. Deswegen brauchen wir globalGS NICHT auf NullHandle() zu prüfen.

```
SYSTEMACTION  ExitHandler
  FreeGS globalGS
END ACTION
```

GetGStringInfo

GetGStringInfo liest die Grafikinformationen eines Graphic String aus. Dazu wird eine Variable vom Strukturtyp **GraphicInfo** belegt. Diese Struktur ist im Abschnitt 2.8.6 (Zeichnen von Bildern) beschrieben.

Syntax: **info** = **GetGStringInfo** (**gsHan**)
info: Variable vom Strukturtyp **GraphicInfo**
gsHan: Handle auf den GString

Beispiel:

```
DIM info as GraphicInfo
DIM gsHan as Handle

gsHan = StartRecordGS()
Rectangle 20, 20, 50, 100
EndRecordGS gsHan

info = GetGStringInfo ( gsHan )
Print "Abmessungen: ";info.sizeX;"x";info.sizeY;"Pixel"

FreeGS gsHan
```

Beispiel: Zeichnen eines GString (in gsHan) zentriert an die Position (50; 100)

```
DIM info as GraphicInfo
DIM x, y
  info = GetGStringInfo ( gsHan )
  x = 50 – info.sizeX/2;
  y = 100 – info.sizeY/2
  DrawGS gsHan, x, y
```

ClipboardGetGS

ClipboardGetGS holt einen GString aus der Zwischenablage. Graphic Strings sind ein universelles Grafikformat unter GEOS. Die Grafik in der Zwischenablage kann z.B. aus GeoWrite, GeoDraw oder dem Sammelalbum kommen. Der GString kann sofort mit DrawGS gezeichnet werden. ClipboardGetGS ist damit analog zur Kombination StartRecordGS / EndRecordGS. Das von ClipboardGetGS gelieferte Handle muss ebenfalls mit FreeGS wieder freigegeben werden.

Syntax: **<han>** = **ClipboardGetGS ( )**

Wird kein GString im Clipboard gefunden so liefert ClipboardGetGS ein NullHandle. ClipboardGetGS setzt die globale Variable clipboardError (Null oder Fehlercode). Sie können vorher mit ClipboardTest prüfen, ob sich ein GString im Clipboard befindet.

Beispiel:

```
DIM gsHan AS HANDLE
  IF ClipboardTest (0, 1) THEN    ! manufID = 0, format = 1
    gsHan = ClipboardGetGS()
    End IF
  <....> z.B. DrawGS gsHan, 0, 0
  FreeGS  gsHan       ' Nicht vergessen
```

ClipboardPutGS

ClipboardPutGS kopiert einen GString in die Zwischenablage. Von dort kann er dann z.B. in GeoDraw eingeklebt werden.

Syntax: **ClipboardPutGS** **<han>**
<han> Handle, das von StartRecordGS geliefert wurde.

Beispiel:

```
DIM gsHan AS HANDLE

  gsHan = StartRecordGS ()
  Rectangle 20, 20, 50, 100, BLACK
  FillEllipse 30, 30, 40, 90, RED
  EndRecordGS gsHan
  ClipboardPutGS gsHan
  FreeGS  gsHan       ' Nicht vergessen
```

SaveGStringAsBackground

SaveGStringAsBackground schreibt einen GString als GEOS Hintergrunddatei. Falls die Datei bereits existiert wird sie überschrieben. WriteGStringToFile setzt die globale Variable fileError (Null oder Fehlercode).

Syntax: **SaveGStringAsBackground** **<han>** , **fileName$**
<han> Handle, das den GString referenziert.
fileName$: Name der anzulegenden Datei (Pfadanteil erlaubt).

ReadGStringFromFile

ReadGStringFromFile liest (kopiert) einen GString aus einer Datei. Aktuell werden nur GEOS Hintergrunddateien unterstützt. ReadGStringFromFile liefert das Handle auf den gelesenen GString. Das Handle muss mit FreeGString wieder freigegeben werden. Die globale Variable fileError wird gesetzt (Null oder Fehlercode). Im Fehlerfall liefert ReadGStringFromFile ein NullHandle.

Syntax: **<han>** = **ReadGStringFromFile** (**fileName$** [, **pictNum**])
<han> Variable vom Typ Handle
fileName$: Name der Datei mit dem Bild (Pfadanteil erlaubt).
pictNum: Nummer des Bildes, falls die Datei mehr als ein Bild enthält.
Wird für GEOS Hintergrunddateien ignoriert.
Default: 1 (erstes Bild lesen)

2.8.6 Zeichnen von Bildern

Dieser Abschnitt beschäftigt sich mit der Ausgabe von fertigen Grafiken, die zur Laufzeit weder erstellt noch verändert werden müssen. Die Ausgabe erfolgt dabei immer in das aktuelle Screen-Objekt. Dabei bezieht sich der Begriff "**Picture**" immer auf Grafiken, die in der Picture-List gespeichert sind (siehe Kapitel 2.8.6.2, Befehle **DrawPicture**, **GetPictureInfo**)). Der Begriff "**Image**" bezieht sich immer auf Grafiken, die in externen Bilddateien (z.B. JPG, PCX, ICO) vorliegen. Dafür stehen die Befehle **DrawImage** und **GetImageInfo** zur Verfügung. Außerdem gibt es das Image-Objekt, das diese Bilder direkt (ohne die Verwendung eines Screen-Objekts) anzeigen kann und im Objekt-Handbuch beschrieben ist.

Die Routinen GetImageInfo, GetPictureInfo GetGStringInfo und GetBitmapInfo ermitteln Informationen über eine Grafik oder eine Grafikdatei in Form der folgenden Struktur:

```
STRUCT GraphicInfo
  sizeX as WORD
  sizeY as WORD
  bitsPerPixel as WORD
  numImages as WORD
End STRUCT
```

Bedeutung der einzelnen Felder:

sizeX und sizeY: Abmessungen der Grafik in Pixeln

bitsPerPixel: Farbtiefe
- 1: Bitmap, monochrom (sw/ws)
- 8: Bitmap, 256 Farben
- 24: Bitmap, True Color
- 0: Graphic String. Farbtiefe je nach Inhalt. Potentiell True Color

numImages: Anzahl der Bilder. Kann bei Dateien (z.B. ICO) größer als 1 sein. Im Fehlerfall: Null.

Tritt beim Aufruf der oben genannten Routinen ein Fehler auf (z.B. Datei nicht gefunden, Bild in der Picture-List nicht vorhanden) gilt:

- numImages ist auf Null gesetzt
- die globale Variable fileError enthält einen Fehlercode. Im Erfolgsfall wird fileError auf Null gesetzt.

Beispiel

```
DIM info as GraphicInfo

info = GetImageInfo (SP_TOP, "GWICON5.ICO")

IF info.numImages = 0 THEN
  Print "Kein Bild gefunden. Fehlercode: ";
                               ErrorText$(fileError)
ELSE
  Print "Abmessungen: ";info.sizeX;"x";info.sizeY;"Pixel"
  Print "Bilder in der Datei:";info.numImages
  IF info.bitsPerPixel = 0 THEN
    Print "Typ: Graphic String"
  ELSE
    Print "Bitmap mit";info.bitsPerPixel;"Bit pro Pixel"
  END IF
END IF
```

2.8.6.1 Zeichnen von Icons

DrawIcon

DrawIcon zeichnet ein Icon aus der Token-Database an die Position (x, y).

Syntax:	**DrawIcon "tchr" , manufID , x , y [, flags]**	
	"tchr":	Tokenchars des Icons. Genau 4 Zeichen
	manufID:	ManufacturerID des Icons. Datentyp WORD
	x:	x-Position der linken oberen Ecke
	y:	y-Position der linken oberen Ecke
	flags:	Icon-Flags. Erlaubte Werte siehe Tabelle Default (kein Flag gesetzt): Standard-Icon zeichnen

Gültige Werte für "flags":

Konstante	Wert	Bedeutung
TOOL_ICON	1	Tool-Icon (15 x 15 Pixel) verwenden.
TINY_ICON	1	Synonym für TOOL_ICON
SMALL_ICON	2	Kleineres Icon verwenden (oft 32x20 Pixel)
BIG_ICON	4	Größeres Icon verwenden (oft 64x40 Pixel)
GRAY_ICON	8	Schwarz-Weiß Icon verwenden
RGB_ICON	16	True-Color Icon verwenden

Wird keines der Flags angegeben wird das "Standard" Icon (meist 48 x 30 Pixel, 16 Farben oder 256 Farben) verwendet.

Hinweise:

- Ist die entsprechend den Flagbits angeforderte Kombination nicht vorhanden sucht das System ein "möglichst passendes" Icon aus. Das Flag "TOOL_ICON" hat dabei Vorrang vor allen anderen Flags.
- Sollte zum gegebenen Token ("TCHR", manufID) kein grafisches Icon vorhanden sein wird ein Text verwendet.
- Findet sich das Token nicht in der TokenDB zeigt R-BASIC ein Ersatzbild ("unbekanntes Icon").
- R-BASIC Icons enthalten nur zwei Bilder: ein Standard- und ein Tool-Icon.

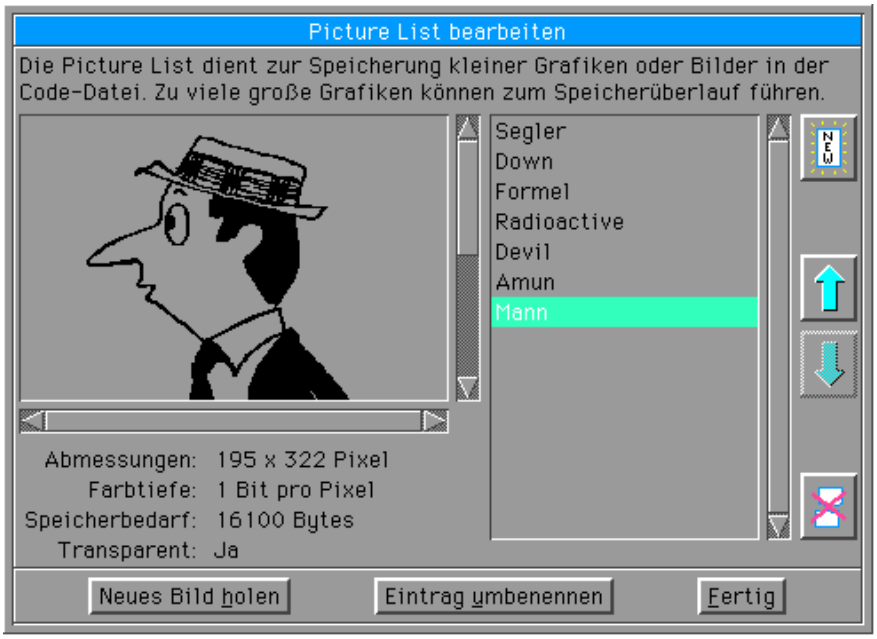
Beispiel: Zeichnen des GeoWrite Document Tool-Icons an die Position (100; 50)

```
DrawIcon "WDAT", 0, 100, 50, TOOL_ICON
```

CaptionIcon

CaptionIcon weist einem Objekt ein Icon aus der Token-Database als grafische "Aufschrift" zu. CaptionIcon ist im Kapitel 3.1 (Die Objektbeschriftung) des Objekthandbuchs beschrieben.

2.8.6.2 Verwendung der "Picture-List"



Die Picture-List ist eine komfortable Möglichkeit Grafiken im der Codedatei selbst unterzubringen und sie zur Laufzeit zu zeichnen. Beim Erstellen Ihres Programms laden Sie über das R-BASIC Menü "Extras" - "Picture-List" Bilder aus einer externen Quelle in die Picture-List. Die Bilder werden dann über ihren Namen angesprochen. Sie können diese Bilder mit dem Befehl **DrawPicture** auf den

Screen zeichnen oder mit der Anweisung **CaptionPicture** (sowohl im UI-Code als auch zur Laufzeit) als grafische Aufschrift für Objekte verwenden.

Als Quellen stehen Ihnen zur Verfügung:
- Externe Bilddateien (z.B. ICO, PCX, JPG).
- Das Clipboard. Insbesondere können Sie über diesen Weg mit GeoDraw selbst gezeichnete Grafiken in Ihr BASIC Programm einbinden.
- Vom Iconeditor "exportierte" Bilder. Der Iconeditor kann Icon-Bilder in eine VM-Datei schreiben (Menü "Icon" - "Schreibe in Datei"), die von der Picture-List eingelesen werden können.

Wenn sie viele, insbesondere größere Bilder in die Picture-List laden sollten Sie der Gesamtgröße ihrer Codedatei etwas Aufmerksamkeit schenken. Erfahrungsgemäß wird GEOS bei Dateigrößen von mehreren 10 Megabyte instabil. Wenn Sie diesbezüglich Probleme haben können Sie Teile ihrer Picture-List in eine Library auslagern.

DrawPicture

DrawPicture zeichnet eine Grafik aus der Picture-List an die Koordinaten x, y.
DrawPicture setzt es die globale Variable fileError - entweder auf Null (das Bild wurde gefunden) oder auf einen Fehlerwert (das Bild wurde nicht gefunden).

Syntax:	**DrawPicture "name" , x , y**	
	"name":	Name des Bildes in der Picture-List
	x:	x-Position der linken oberen Ecke
	y:	y-Position der linken oberen Ecke

Wenn DrawPicture im Code einer Library gerufen wird bezieht sich der Name des Bildes auf die Picture-List der Library. Das ermöglicht es unter anderem Bilder in die Picture-List von Libraries auszulagern.

Beispiel: Zeichnen eines Bildes an die Position (50; 100)

```
DrawPicture "Mann", 50, 100
```

Tipp: Wenn der Compiler den Namen des Bildes in der Picture-List ermitteln kann (d.h. der Name steht wie im Beispiel im Klartext da und wird nicht durch den Aufruf von Stringfunktionen wie Left$ "berechnet") wird der Name sofort in die interne Nummer des Bildes umgerechnet. Damit muss das Bild zur Laufzeit nicht mehr gesucht werden und es wird viel schneller gezeichnet.

GetPictureInfo

GetPictureInfo liest die Grafikinformationen eines Bildes aus der Picture-List aus. GetPictureInfo setzt die globale Variable fileError (Null oder Fehlercode).

Syntax: **info** = **GetPictureInfo** (**"name"**)
info: Variable vom Strukturtyp **GraphicInfo**
"name": Name der Grafik in der Picture-List

Beispiel:

```
DIM info as GraphicInfo
info = GetPictureInfo ( "Down Arrow" )
Print "Abmessungen: ";info.sizeX;"x";info.sizeY;"Pixel"
```

Beispiel: Zeichnen eines Bildes zentriert an die Position (50; 100)

```
DIM info as GraphicInfo
DIM x, y
  info = GetPictureInfo ( "Down Arrow" )
  x = 50 – info.sizeX/2;
  y = 100 – info.sizeY/2
  DrawPicture "Down Arrow", x, y
```

CaptionPicture

CaptionPicture weist einem Objekt ein Bild aus der Picture-List als grafische "Aufschrift" zu. CaptionPicture ist im Kapitel 3.1 (Die Objektbeschriftung) des Objekthandbuchs beschrieben.

2.8.6.3 Externe Bilddateien

Sie können Bilder aus externen Bilddateien zur Laufzeit direkt auf den Screen von R-BASIC zeichnen. Die Routine **DrawImage** übernimmt dabei alle notwendigen Schritte, vom Öffnen der Datei über das Einlesen und Konvertieren in eine GEOS-kompatibles Format bis zum Zeichnen auf den Schirm und das abschließende Schließen der Bilddatei.

DrawImage

DrawImage zeichnet eine Grafik. Die Grafik wird aus einer externen Datei gelesen. Sollte die Datei mehr als ein Bild enthalten (z.B. *.GIF, *.ICO) können Sie mit dem Parameter pictNum bestimmen, welches Bild ausgelesen wird. Das erste Bild hat immer die Nummer Null.

Syntax:	**DrawImage [stdPath,] , "Path+File" , x y , [, pictNum]**
	stdPath: Optional: Standardpfad Konstante, z.B. SP_TOP
	"Path+File": Dateiname, Pfade sind zulässig
	x: x-Position der linken oberen Ecke
	y: y-Position der linken oberen Ecke
	pictNum: Optional: Nummer des Bildes in der Datei

- Wird kein Standardpfad angegeben wird die Datei im aktuellen Verzeichnis gesucht.
- Unterstützte Dateiformate: JPG, BMP, ICO, PCX, GIF, TGA, RLE, DIB, SCR (BreadBox SplashScreen), FLC, FLI sowie GEOS Hintergrunddateien
- Die externe Datei wird zur Laufzeit geöffnet, d.h. sie **muss unbedingt** in das R-App Paket aufgenommen werden oder es muss auf andere Weise sichergestellt sein, dass sie existiert.
- Wird pictNum nicht angegeben so wird immer das erste Bild ausgelesen.
- Die globale Variable fileError wird gesetzt - entweder auf Null (das Bild wurde gefunden) oder auf einen Fehlerwert (die Datei wurde nicht gefunden oder sie enthält kein Bild).

Beispiele:

```
DrawImage  SP_TOP, "GWICON5.ICO", 0, 0
DrawImage  "BILDER\\SUNSET.JPG", 100, 100
DrawImage  SP_DOCUMENT, "R-BASIC\\BILD2.PCX" 0, 20
```

GetImageInfo

GetImageInfo liest die Grafikinformationen aus einer Datei aus. Wird kein Standardpfad angegeben wird die Datei im aktuellen Verzeichnis gesucht. Der Dateiname darf einen Pfadanteil enthalten. GetImageInfo setzt die globale Variable fileError (Null oder Fehlercode).
Unterstützte Dateiformate: Siehe DrawImage

Syntax: **info** = **GetImageInfo** ([**stdPath**,] **"Path+File"**)
info: Variable vom Strukturtyp **GraphicInfo**
stdPath: Optional: Standardpfad Konstante, z.B. SP_TOP
"Path+File": Dateiname, Pfade sind zulässig

Beispiele:

```
DIM info as GraphicInfo

info = GetImageInfo (SP_TOP, "GWICON5.ICO")
info = GetImageInfo ("BILDER\\SUNSET.JPG")
info = GetImageInfo (SP_DOCUMENT, "R-BASIC\\BILD2.PCX")
```

Beispiel: Zeichnen eines Bildes zentriert an die Position (50; 100)

```
DIM info as GraphicInfo
DIM x, y
  info = GetImageInfo ( "BILDER\\SUNSET.JPG" )
  x = 50 - info.sizeX/2;
  y = 100 - info.sizeY/2
  DrawImage "BILDER\\SUNSET.JPG", x, y
```

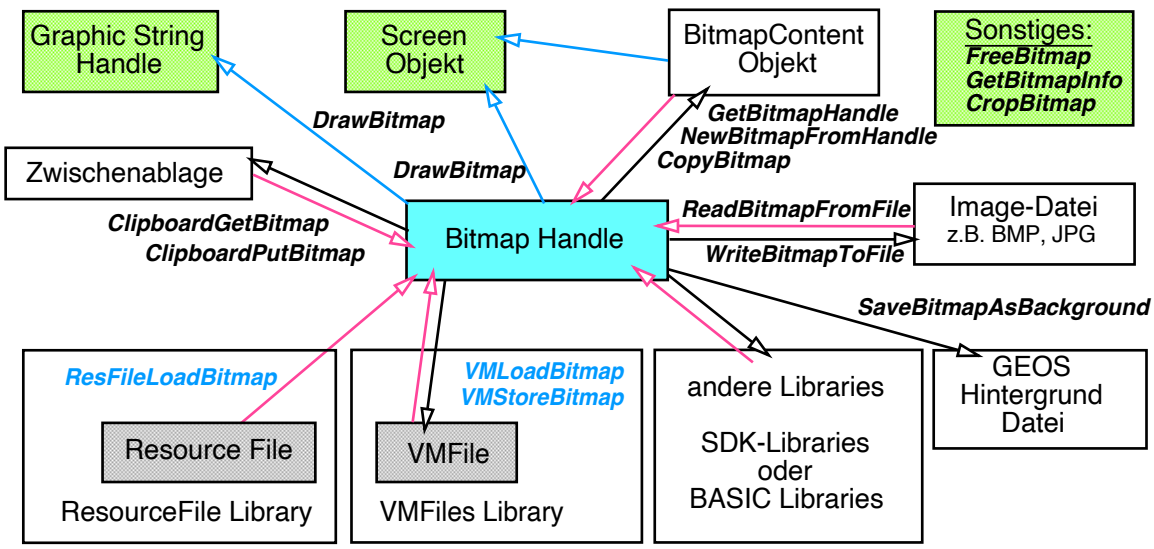
CaptionImage

CaptionImage weist einem Objekt ein Bild aus einer externen Bilddatei als grafische "Aufschrift" zu. CaptionImage ist im Kapitel 3.1 (Die Objektbeschriftung) des Objekthandbuchs beschrieben.

2.8.6.4 Bitmaps und Bitmap Handles

Bitmaps werden üblicher Weise von einem BitmapContent verwaltet. Die meisten Informationen zu Bitmaps finden Sie daher auch bei der Beschreibung des BitmapContent-Objekts im Objekthandbuch. Es gibt jedoch auch die Möglichkeit Bitmaps über ein Handle anzusprechen. Mit dieser Thematik beschäftigt sich dieses Kapitel.

Das Konzept der Bitmap Handles (eine Variable vom Typ HANDLE, die eine Bitmap referenziert) dient zum Austausch einer Bitmap-Grafik zwischen verschiedenen Komponenten eines BASIC Programms. Insbesondere kann eine Bitmap mit der Routine DrawBitmap in andere Objekte, z.B. eine andere Bitmap, ein beliebiges anders Screen-Objekt oder in einen Graphic String gezeichnet werden. Einige R-BASIC Libraries bieten die Möglichkeit Bitmaps in einer externen Datendatei zu speichern oder von dort zu laden. Das folgende Bild gibt eine Übersicht über die Möglichkeiten.



Wie im Bild zu sehen gibt es sechs Möglichkeiten an ein Bitmap-Handle zu kommen:

1. Das BitmapContent-Objekt stellt die Methode GetBitmapHandle bereit. Diese Methode liefert ein Handle, das direkt die Bitmap im BitmapContent-Objekt referenziert, das heißt die Bitmap wird nicht kopiert! Sie **dürfen** das Handle **nicht** mit der Routine FreeBitmap **freigeben**!
2. Die Routine CropBitmap (siehe Kästchen "Sonstiges" im Bild oben) kopiert einen Ausschnitt oder die gesamte Bitmap in eine neue.
3. Die Methode CopyBitmap kopiert die Bitmap eines BitmapContent Objekts und liefert das Handle auf die Kopie. Sie müssen die Kopie mit der Routine FreeBitmap wieder freigeben!
4. Die Routine ClipboardGetBitmap legt eine Kopie einer im Clipboard befindlichen Bitmap an und liefert das Handle auf diese Kopie. Sie müssen eine so angelegte Bitmap mit der Routine FreeBitmap wieder freigeben!

5. Die Routine ReadBitmapFromFile liest eine Bitmap aus einer Bilddatei. Sie müssen eine so angelegte Bitmap mit der Routine FreeBitmap wieder freigeben!
6. Einige Libraries sind in der Lage Bitmaps in einer Datei zu speichern. Diese Libraries liefern Routinen mit, um auf die Bitmaps in der Datei zuzugreifen. Im Bild oben sind die VMFiles Library (VMLoadBitmap) und die ResourceFile Library (ResFileLoadBitmap) angegeben. Auch diese Routinen liefern eine Referenz auf die in der Datei befindliche Bitmap, die Bitmap wird nicht kopiert.

DrawBitmap

DrawBitmap zeichnet den Bitmap Grafik an die Position x, y.

Syntax: **DrawBitmap** **<han>** , **x** , **y** [, **noFix**]

<han>:	Handle, das die Bitmap referenziert.
x, y:	Zeichenposition in Pixeln. Die linke obere Ecke der Bitmap wird an diese Position gezeichnet.
noFix:	FALSE: 8 Bit Bitmap fix anwenden (default) TRUE: 8 Bit Bitmap nicht fix anwenden (siehe unten)

Bugs ...

Das GEOS System (mindestens bis Version 4.1.3) crasht wenn folgende Bedingungen **gleichzeitig** zutreffen:

- der Screen ist eine 8 Bit Bitmap (mit oder ohne Transparenz)
- und es soll eine Bitmap mit Maske (monochrome, 8 Bit oder 24 Bit) gezeichnet werden
- und die x-Koordinate ist negativ

R-BASIC löst dieses Problem, indem es im oben genannten Fall die zu zeichnende Bitmap vorher beschneidet und dann so zeichnet, als gäbe es dieses Problem nicht.

Wenn Sie den Parameter noFix (TRUE) angeben, wendet R-BASIC diesen Fix nicht an. Das kann z.B. in folgenden Situationen sinnvoll sein:

- Sie haben ein verschobenes Koordinatensystem (siehe Anweisung Screen-SetTranslation) und sind sicher, dass die Bitmap trotz negativer x-Koordinate nicht über den linken Rand hinausragt.
- Sie haben eine neuere GEOS-Version, die den Bug gefixt hat.

... und Features?

Wenn Sie mit DrawBitmap eine monochrome Bitmap in eine andere Bitmap drawen ist das Ergebnis etwas seltsam. Je nach Situation wird die Bitmap beispielsweise transparent gezeichnet, obwohl sie nicht gar keine Maske hat oder die Bitmapdaten landen in der Maske der Zielbitmap. Ist das nun ein Bug oder ein Feature?

CropBitmap

CropBitmap (engl. to crop: etwas zuschneiden) kopiert einen Ausschnitt einer Bitmap. Der Ausschnitt wird durch die übergebenen Koordinaten bestimmt. Es ist zulässig, dass die Koordinaten einen Bereich beschreiben, der teilweise außerhalb der Bitmap liegt. CropBitmap handelt alle "Koordinatenfehler" korrekt.
CropBitmap liefert ein Handle auf die Kopie. Dieses Handle muss mit FreeBitmap wieder freigegeben werden. Im Fehlerfall (der durch die angegeben Koordinaten Ausschnitt liegt komplett außerhalb der Bitmap) liefert CropBitmap ein NullHandle.
CropBitmap kann verwendet werden, um die ganze Bitmap zu kopieren. Geben Sie dazu einen Ausschnitt an, der sicher größer ist, als die zu kopierende Bitmap.

Syntax:	**<han>** = **CropBitmapInfo** (**bmpHan** , **x0**, **y0**, **x1**, **y1**)
	bmpHan: Handle auf die vorhandene Bitmap
	x0, y0: linke obere Ecke des zu kopierenden Ausschnitts
	x1, y1: rechte untere Ecke des zu kopierenden Ausschnitts

Die Größe der neuen Bitmap ist (falls die Koordinaten nicht außerhalb der Bitmap liegen):

xSize = x1 - x0 + 1
ySize = y1 - y0 + 1

Liegt ein Koordinatenpaar außerhalb der Bitmap so ist der kopierte Ausschnitt kleiner.

Beispiele:

```
' In den folgenden Beispielen gilt:
DIM bmpHan, newHan as HANDLE
' bmpHan soll eine Bitmap referenzieren
```

```
  ' Kopieren eines 50 x 150 Pixel großen Ausschnitts
  newHan = CropBitmap ( bmphan, 0, 0, 49, 150 )
```

```
  ' Negative Koordinaten sind zulässig
  ' Der kopierte Ausschnitt ist 64 x 16 Pixel groß
  newHan = CropBitmap ( bmpHan, -7, -100, 63, 15 )
```

```
  ' Kopieren der kompletten Bitmap.
  ' x1 und y1 liegen sicher außerhalb der Bitmap
  newHan = CropBitmap ( bmpHan, 0, 0, 1E6, 1E6 )
```

GetBitmapInfo

GetBitmapInfo liest die Grafikinformationen einer Bitmap, die durch ein Bitmap-Handle referenziert wird, aus. Die Struktur GraphicInfo wurde weiter oben beschrieben.

Syntax: **info** = **GetBitmapInfo** (**bmpHan**)
info: Variable vom Strukturtyp **GraphicInfo**
bmpHan: Handle auf die Bitmap

Beispiel:

```
DIM info as GraphicInfo
DIM bmpHan as Handle

bmpHan = BitmapObj.GetBitmapHandle  ' Bitmap wird nicht kopiert

info = GetBitmapInfo ( bmpHan )
Print "Abmessungen: ";info.sizeX;"x";info.sizeY;"Pixel"
```

ReadBitmapFromFile

ReadBitmapFromFile liest eine Bitmapgrafik aus einer Datei. Die Datei wird geöffnet, die Bilddaten werden kopiert und anschließend wird die Datei wieder geschlossen. ReadBitmapFromFile liefert das Handle auf die gelesene Bitmap. Das Handle muss mit FreeBitmap wieder freigegeben werden. Die globale Variable fileError wird gesetzt (Null oder Fehlercode). Im Fehlerfall liefert ReadBitmapFromFile ein NullHandle.

Syntax: **<han>** = ReadBitmapFromFile (**fileName$** [, **pictNum**])
<han> Variable vom Typ Handle
fileName$: Name der Datei mit dem Bild (Pfadanteil erlaubt).
pictNum: Nummer des Bildes, falls die Datei mehr als ein Bild enthält.
Default: 1 (erstes Bild lesen)

ReadBitmapFromFile unterstützt die folgenden Dateiformate: BMP, RLE, DIB, ICO, PCX, GIF, FLI, FLC, JPG, TGA, SCR (BreadBox SplashScreen).
Tipp: Mit Routine ReadGStringFromFile können Sie das Bild aus einer GEOS Hintergrunddatei auslesen.
Tipp: Die Routine GetImageInfo liefert detaillierte Informationen über die Bilddatei.

Der folgende Code liest eine PCX-Datei und nutzt ein BitmapContent-Objekt um die Grafik anzuzeigen. Das BitmapContent Objekt legt sich eine Kopie der Bilddaten an, so dass wir das Handle mit FreeBitmap wieder freigeben können.

```
SUB LoadAndShowImage()
DIM h as HANDLE
  h = ReadBitmapFromFile ( "WOLKEN.PCX" )
  IF fileError THEN RETURN
  MyBitmapContent.NewBitmapFromHandle h    ' Kopiert die Daten
  FreeBitmap h                ' gibt die Bitmapdaten wieder frei.
End SUB
```

WriteBitmapToFile

WriteBitmapToFile schreibt eine Bitmap, die durch ein Handle referenziert wird, im BMP-Format in eine Datei. Der Dateiname sollte deshalb auf BMP enden. Falls die Datei bereits existiert wird sie überschrieben. WriteBitmapToFile setzt die globale Variable fileError (Null oder Fehlercode).

Syntax: **WriteBitmapToFile** **<han>** , **fileName$**
<han> Handle, das eine Bitmap referenziert.
fileName$: Name der anzulegenden Datei (Pfadanteil erlaubt).

WriteBitmapToFile berücksichtigt eine eventuell vorhandene Maske. Transparente Pixel werden auf die Farbe Weiß bzw. auf den Index 255 (das entspricht in der Standardpalette ebenfalls Weiß) gesetzt. Eine vorhandene Palette wird ebenfalls berücksichtigt.
Der folgende Code schreibt die Bitmap eines BitmapContent Objekts in eine Datei. Weil die Methode GetBitmapHandle nur das Handle liefert, die Daten aber nicht kopiert dürfen wir das Handle NICHT mit FreeBitmap freigeben.

```
SUB WriteImageToFile ( )
DIM h as Handle
  h = MyBitmapContent.GetBitmapHandle
  WriteBitmapToFile h, "BILD1.BMP"
End SUB
```

SaveBitmapAsBackground

SaveBitmapAsBackground schreibt eine Bitmap, die durch ein Handle referenziert wird, als GEOS Hintergrunddatei. Falls die Datei bereits existiert wird sie überschrieben. WriteBitmapAsBackground setzt die globale Variable fileError (Null oder Fehlercode).

Syntax: **SaveBitmapAsBackground** **<han>** , **fileName$**
<han> Handle, das eine Bitmap referenziert.
fileName$: Name der anzulegenden Datei (Pfadanteil erlaubt).

FreeBitmap

FreeBitmap gibt das Handle und die komplette Bitmap wieder frei. Der von Bitmap belegte Speicherplatz wird freigegeben. FreeBitmap darf nur auf Handles angewendet werden, von ClipboardGetBitmap, ReadBitmapFromFile, CropBitmap oder der Methode CopyBitmap belegt wurden!

Syntax: **FreeBitmap** **<han>**
<han> Handle, das die Bitmap referenziert.

Tipp: Um sicherzustellen, dass das System nicht crasht, falls das Handle irrtümlich noch einmal mit DrawBitmap verwendet wird, können Sie es nach dem Freigeben durch FreeBitmap () mit der Anweisung "han = NullHandle()" löschen. Draw-Bitmap gibt dann nur eine entsprechende Fehlermeldung aus.

ClipboardGetBitmap

ClipboardGetBitmap holt eine Bitmap aus der Zwischenablage. Die Bitmap kann sofort mit DrawBitmap gezeichnet werden. Das von ClipboardGetBitmap gelieferte Handle muss mit FreeBitmap wieder freigegeben werden.

Syntax: **<han>** = **ClipboardGetBitmap ( )**

Wird keine Bitmap im Clipboard gefunden so liefert ClipboardGetBitmap ein NullHandle. ClipboardGetBitmap setzt die globale Variable clipboardError (Null oder Fehlercode). Sie können vorher mit ClipboardTest prüfen, ob sich eine Bitmap im Clipboard befindet.

Beispiel:

```
DIM bmpHan AS HANDLE

  IF ClipboardTest (0, 7) THEN    ! manufID = 0, format = 7
    bmpHan = ClipboardGetBitmap()
    End IF

    <....>
    z.B. DrawBitmap bmpHan, 0, 0

  FreeBitmap  bmpHan      ' Nicht vergessen
```

ClipboardPutBitmap

ClipboardPutBitmap kopiert eine Bitmap, die durch ein Bitmap-Handle referenziert wird, in die Zwischenablage. Von dort kann sie in andere Applikationen eingeklebt oder von anderen R-BASIC Objekten gelesen werden.

Syntax: **ClipboardPutBitmap <han>**
<han> Handle, das eine Bitmap referenziert.

Beachten Sie, dass Sie eine Bitmap, die sich in einem BitmapContent-Objekt befindet, auch direkt mit der Methode ClpCopy in die Zwischenablage kopiert werden kann.
Beispiel. Wir nehmen an, dass die Routine FindABitmapHandle ein Bitmap-Handle zurückgibt.

```
DIM bmpHan AS HANDLE
  bmpHan = FindABitmapHandle()
  ClipboardPutBitmap bmpHan
```

(Leerseite)