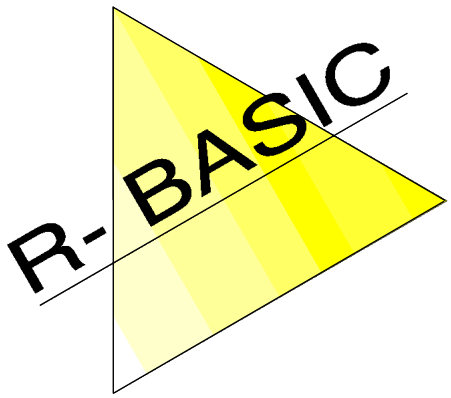
R-BASIC

Einfach unter PC/GEOS programmieren



Benutzer-Handbuch

Volume 2
Übersetzung von Programmen, Debugging

Version 1.0

(Leerseite)

Inhaltsverzeichnis

4 Übersetzen von BASIC-Programmen 44

5 Fehlersuche und Debugging 48

5.1 Methoden zur Fehlersuche 48
5.2 Die Error-Checking Version 52
5.3 Der R-BASIC Debugger 55

5.3.1 Bedienung des Debuggers 56
5.3.2 Debuggen von Libraries 61
5.3.3 Interne Organisation 61

(Leerseite)

4 Übersetzen von R-BASIC Programmen

R-BASIC unterstützt das Übertragen von Programmen in eine andere Sprache sehr komfortabel. Der Einsteiger mag es für eine gute Idee halten, den Quellcode durchzugehen, jeden Textstring zu ändern und dann das Programm neu zu compilieren. Das wäre jedoch nicht nur mühsam und fehleranfällig, sondern müsste nach jeder noch so kleinen Änderung des Programms wiederholt werden.

Deswegen wird mit R-BASIC ein "Übersetzer" Tool (R-BASIC Translator) mitgeliefert. Dieses Tool öffnet ein fertig compiliertes R-BASIC-Programm, liest die zu übersetzenden Strings aus, ermöglicht deren Übersetzung und speichert eine übersetzte Kopie des Originalprogramms. Die Übersetzungsdaten, d.h. die Originaltexte und die übersetzten Texte, werden außerdem in einer eigenen Datei (im Folgenden "Übersetzungsdatei" genannt) abgelegt, so dass sie wieder-verwendet werden können, wenn später eine neue Version des Programms übersetzt werden soll.

Ein weiterer Vorteil dieses Prinzips ist es, dass die Übersetzung nicht direkt vom Programmierer ausgeführt werden muss. Vielmehr kann auch ein Sprachkundiger, der keine Ahnung vom Programmieren hat, nach kurzer Einweisung Ihr Programm übersetzen.

Da in den meisten Fällen die Originalversion und die übersetzte Version den gleichen Namen haben und auch am gleichen Ort gespeichert werden sollen bzw. müssen (z.B. bei Libraries), ist es erforderlich, die übersetzte Version in einer zweiten PC/GEOS-Installation, dem sogenannten Target (engl. target = Ziel), abzulegen. Dieses Target müssen Sie vorher anlegen und den Pfad dorthin dem "Übersetzer" einmalig mitteilen. Wenn Sie alle zu Ihrem Projekt gehörenden Dateien in die Target-Installation kopieren können Sie das Installationspaket für die übersetzte Version komplett unter dem Target anlegen.

Um ein R-BASIC-Programm zu übersetzten müssen Sie also folgendermaßen vorgehen:

1. Stellen Sie sicher, dass eine lauffähige Target-Installation existiert.

2. Compilieren Sie ihr Programm. Legen Sie eine ein eigenständig lauffähiges Programm (R-App) an. Nur diese können übersetzt werden.

3. Öffnen Sie das zu übersetzende Programm mit dem R-BASIC Translator. Dazu gehen Sie folgendermaßen vor:

   Wenn Sie das Programm erstmalig übersetzen:
   - Starten Sie das Übersetzer-Tool direkt. Wählen Sie "BASIC Programm" und öffnen damit den Launcher des BASIC-Programms im World-Verzeichnis.

   Falls bereits eine Übersetzungsdatei existiert:
   - Öffnen Sie die Übersetzungsdatei bitte diese durch Doppelklick.
   - Wählen Sie "Übersetzungsdatei updaten" aus dem Dateimenü. Der Translator vergleicht damit die in der Übersetzungsdatei gespeicherten

Informationen mit der aktuellen Version und weiß jetzt, ob zu übersetzende Texte geändert wurden, hinzugekommen sind oder gelöscht wurden. Sie bekommen die entsprechenden Informationen angezeigt.

4. Übersetzen Sie alle Textstrings und speichern die Übersetzer-Daten als Dokument des Übersetzer-Tools (Übersetzungsdatei).

5. Erzeugen Sie das übersetzte R-BASIC-Programm. Wählen Sie dazu den Menüpunkt "Übersetztes Programm anlegen" aus dem Dateimenü. Der R-BASIC Translator kopiert alle notwendigen Dateien ins Target und passt die übersetzten Strings an.

6. Wechseln Sie ins Target und testen Ihr übersetztes Programm. Achten Sie dabei besonders auf die Tastenkürzel zur Tastaturnavigation.

7. Erzeugen Sie ein Installationspaket für das übersetzte Programm.

Eine ausführliche Beschreibung, wie man den R-BASIC Translator bedient finden Sie in der Hilfe zum Programm. Dort sind auch die Terminologie und weitere Details ausführlich erläutert.

Anweisungen an das Übersetzer-Tool

Anweisung	Syntax im UI-Code und im BASIC-Code
NoTranslate	NoTranslate
TranslateLen	TranslateLen **<n>**

In bestimmten Situationen ist es gar nicht gewünscht, wenn bestimmte Strings durch einen Übersetzer geändert werden können. Wenn Sie z.B. ein Spiel programmieren, dessen Levels durch Strings in DATA-Zeilen beschrieben werden, müssen diese Level-Strings vor versehentlichen Änderungen durch den Übersetzer geschützt werden. Ein häufiger Fall sind auch die Zugriffs-Flags oder Dateinamen bei Datei-Operationen (z.B. FileOpen), die als Strings im Quelltext stehen.
Ein anderes potentielles Problem stellen Strings dar, die eine bestimmte Länge nicht überschreiten dürfen.
Zur Lösung dieser Probleme bietet R-BASIC die Steueranweisungen **NoTranslate** und **TranslateLen** an. Diese können überall im Code, sowohl im BASIC-Code als auch im UI-Code, stehen. Sie steuern das Anlegen der Übersetzungs-informationen, werden aber selbst nicht in im compilierten Code abgespeichert.

NoTranslate

Syntax im Code **NoTranslate**

Die Anweisung **NoTranslate** bewirkt, dass alle Textstrings, die in der darauf-folgenden Anweisung (bis zum Zeilenende oder dem folgenden Doppelpunkt) stehen, vom Übersetzter Tool nicht geändert werden können. In allen drei folgenden Code-Beispielen können die Strings "Ich bin" und "Jahre alt." nicht übersetzt werden. Der String "Das ist viel" ist wieder übersetzbar.

```
NoTranslate
Print "Ich bin"; n; "Jahre alt."
Print "Das ist viel."
```

```
NoTranslate : Print "Ich bin"; n; "Jahre alt."
Print "Das ist viel."
```

```
NoTranslate
Print "Ich bin"; n; "Jahre alt." : Print "Das ist viel."
```

Weitere Beispiele:

```
NoTranslate
DATA "Ein", "Mops", "kam"     ' alle nicht übersetzbar
DATA "in", "die", "Küche"     ' wieder übersetzbar
```

```
NoTranslate
FileCreate "info.txt", "orw"
```

Beispiel im UI-Code

```
Button  ApplyButton
  NoTranslate
  Caption$ = "Anwenden", 2
End Object
```

TranslateLen

Syntax im Code **TranslateLen** **n**
n: numerischer Wert - maximale Textlänge bei Übersetzung

Die Anweisung "**TranslateLen** n" bewirkt, dass allen Textstrings, die in der darauf folgenden Anweisung (bis zum Zeilenende oder dem folgenden Doppelpunkt)

stehen, vom Übersetzer-Tool nur bis zu n Zeichen zugewiesen werden können. Der Standardwert für die maximale Länge übersetzter Texte ist 1024, der Maximalwert ist ebenfalls 1024.
"TranslateLen 0" ist identisch mit "NoTranslate".

Beispiel:

```
TranslateLen 32
CONST tempFile$ = "MyTmp GEOS File"    ' max. 32 Zeichen
CONST info$ = "Sind Sie sicher?"  ' wieder frei übersetzbar
```

5 Fehlersuche und Debugging

5.1 Methoden zur Fehlersuche

Um einen Fehler einzugrenzen gibt es keinen allgemeingültigen Algorithmus. Es gibt jedoch einige Strategien, sie sich bewährt haben.

Hilfreiche Befehle
Die in der folgenden Tabelle aufgelisteten Befehle und globale Variablen können Ihnen bei der Fehlersuche behilflich sein.

Befehl	Bedeutung
fileError	Globale Variable. Enthält die Information, ob die letzte Dateioperation erfolgreich war oder nicht.
clipboardError	Globale Variable. Enthält die Information, ob die letzte Clipboardoperation erfolgreich war oder nicht.
customError	Globale Variable zur freien Verfügung des Programmierers. Üblicher Weise verwendet um Programmspezifische Fehlercodes abzuspeichern.
ErrorText$	String-Funktion, die einen System-Fehlercode in eine "verständliche" Form übersetzt.
HandleInfo$	String-Funktion, die interne Informationen über ein Handle liefert.
FileInfo$	String-Funktion, die interne Informationen über eine Datei-Variable liefert.
ObjInfo$	String-Funktion, die interne Informationen über ein Objekt liefert.
MsgBox, ErrorBox	Geben eine Information in einer Dialogbox aus.
QuestionBox	Gibt eine Information in einer Dialogbox aus und erwartet eine Entscheidung vom Nutzer.
EC, NC, ECON, ECOFF	Steuern, ob eine bestimmte Codezeile compiliert wird oder nicht.
WaitForHandles	Sicherstellen, dass die Anzahl freier Handles nicht zu niedrig ist (siehe unten).
IgnoreWarning	Nächste Compilerwarnung ignorieren.

Defensive Programmierung
Rechnen Sie jederzeit damit, dass etwas schief geht, auch wenn Sie der Meinung sind, dass gar nichts schief gehen kann. Beispiele:

- Ein typisches Problemfeld sind Dateioperationen. Prüfen sie z.B. nach dem Öffnen einer Datei immer die globale Variable **fileError** ab, auch wenn Sie sicher sind, dass die Datei existieren muss. Gerade in der Entwicklungsphase von Programmen kommt es z.B. vor, dass Sie irgendwo das Verzeichnis gewechselt haben und deswegen die Datei im aktuellen Verzeichnis doch nicht zu finden ist.

- Verwenden Sie in jeder ON ... SWITCH Anweisung einen DEFAULT-Zweig, in dem Sie z.B. mit einer MsgBox darüber informiert, dass ein bestimmter Fall nicht behandelt wurde.

Nachteil einer extensiv defensiven Programmierung ist, dass sich das Programm aufblähen und / oder verlangsamen kann.

Ausgabe von Log-Informationen

Um zu überprüfen, ob ein bestimmter Programmteil auch genau so arbeitet, wie man es wünscht, kann man sich in bestimmten Abständen eine Information ausgeben lassen, dass dieser Programmteil gerade erfolgreich abgearbeitet wurde - oder eben nicht. Im einfachsten Fall verwendet man dazu den Befehl **MsgBox**, eventuell in Kombination mit dem Befehl **ErrorText$**. Die komfortable Variante sieht so aus, dass man sich ein eigenes Objekt zur Ausgabe von Log- oder Fehlermeldungen definiert (z.B. ein Memo oder ein BitmapContent) und sich eine Routine schreibt, die Text an dieses Objekt ausgibt. Auf diese Weise kann man auch den Zustand von bestimmten Variablen ausgeben. Für ein BitmapContent als Ausgabeobjekt könnte diese Routine so aussehen:

```
SUB DebugLog(text$ as String)
DIM scr as Object
  scr = Screen   ' aktuellen Screen merken
  Screen = MyBitmapContent
  Print text$
  Screen = scr   ' Screen zurücksetzen
End Sub
```

Wenn Sie ein Memo Objekt zur Ausgabe der Informationen verwenden müssen Sie die Print-Anweisung ersetzen und können sich das Umschalten des Screens ersparen. In einem Memo können Sie auch mehr Text anzeigen, Sie müssen jedoch selbst dafür sorgen, dass das Fassungsvermögen des Objekts nicht überschritten wird.

Verwenden der Error-Checking Version

R-BASIC bietet die Möglichkeit bestimmte Codezeilen nur dann zu Compilieren, wenn man eine "Error-Checking Version" (Version zur Entwicklung bzw. Fehlersuche) compiliert. Zwischen dieser und der "normalen" Version kann mit wenigen Mausklicks umgeschaltet werden. In Kombination mit den oben genannten Methoden kann man damit eine extensive Fehlerkontrolle implementieren, ohne die damit verbundenen Nachteile in der Endversion des Programms zu haben. Die Verwendung von Error-Checking Code wird ausführlich im nächsten Kapitel beschrieben.

Verwenden des Debuggers

Der Debugger ist die leistungsfähigste Methode der Fehlersuche. Sie können damit zum Beispiel Ihren Code schrittweise abarbeiten, Variablen ansehen und ändern, überprüfen in welcher Reihenfolge Routinen aufgerufen werden und vieles mehr. Es wird dringend empfohlen sich mit der Arbeitsweise des Debuggers

vertraut zu machen. Auch wenn er anfangs relativ komplex erscheint ist er doch sehr einfach und intuitiv zu bedienen. Der Debugger ist ausführlich im Kapitel 5.3 beschrieben.

WaitForHandles

Unter GEOS ist es ein häufiges Problem, dass die Systemhandles knapp werden. Unglücklicher Weise verbrauchen bestimmte Operationen, insbesondere einige Dateioperationen oder das Setzen bestimmter Instancevariablen, temporär einige Handles. Führt man diese Operationen oft hintereinander aus, z.B. in einer Schleife, so kann es zum berüchtigten "zu wenig Handles" Fehler kommen. Das ist ein Problem auf Systemebene, tritt also auch bei reinen SDK-Programmen auf. R-BASIC bieten einen "WorkAround" für dieses Problem. Falls bei Ihrem Programm ein solches Problem auftritt können Sie den Befehl **WaitForHandles** verwenden. Dieser WorkAround wird z.B. auch im Uni-Installer Programm verwendet.

R-BASIC Programme merken sich die Anzahl der freien Handles am Programmstart. WaitForHandles stoppt die Ausführung eines BASIC Programms für eine bestimmte Zeit, wenn die Anzahl der verfügbaren Handles unter einem bestimmten Wert liegt. Während dessen hat das System Zeit, die temporär benutzten Handles freizugeben, so dass das BASIC Programm anschließend problemlos weiterarbeiten kann.

Syntax: WaitForHandles [p [, t [, n]]]

- p: Prozentualer Anteil, unter den die Anzahl an frei verfügbaren Handles sinken muss, damit das BASIC Programm pausiert.
  Default: 25
- t: Zeit in Tics (1 tic = 1/60s), die das BASIC Programm warten soll.
  Default: 6 (1/10s)
- n: Anzahl der Versuche, die 't' Tics gewartet werden soll, bis die Anzahl an freien Handles wieder größer ist. Wird die Anzahl 'n' überschritten arbeitet das R-BASIC Programm weiter.
  Default: 20 (-> per Default insgesamt 2 Sekunden warten)

  Wenn Sie für p, t oder n den Wert Null angeben wird der Defaultwert benutzt.

Beispiele

```
WaitForHandles              ' Defaultwerte verwenden
WaitForHandles 50, 6        ' Bei weniger als 50% der Handles
                            ' 0,1 Sekunde warten
```

Ein typisches Vorgehen, dieses Problem unter R-BASIC hervorzurufen, ist das sehr häufig Aufrufen einer Sequenz, in der eine Datei geöffnet, bearbeitet und wieder geschlossen wird. Für den Fall, dass man eine Datei benutzt um die Schritte eines Programms zu dokumentieren, ist das ein durchaus realistisches

Szenario. Die folgende Schleife ruft den "zu wenig Handles" Fehler hervor, falls der Befehle WaitForHandles entfernt wird.

```
FOR n = 10 TO 1000
  FOR c = Asc("A") TO Asc("z")
    fh = FileOpen "crash.txt"
    FileWrite fh, c, 1
    FileClose fh
    WaitForHandles
    NEXT
NEXT
```

Ein anderes Szenario ist das schnelle und periodische Setzen einer Instance-variablen, die ein visuelles Update eines oder mehrerer Objekte hervorruft. Diese Objekte senden dann wieder Messages an andere Objekte usw. Da das Senden von Messages temporär Handles erfordert kann das in seltenen Fällen zu einem Problem führen.
In allen Fällen gilt aber auch: ob das Problem auftritt oder nicht kann auch von der Systemgeschwindigkeit abhängen.

IgnoreWarning

Der Compiler gibt in verschiedenen Situationen eine Warnung aus, um Sie auf potentiell fehlerhaften Code hinzuweisen. Ein typisches Beispiel ist der Vergleich von Strukturvariablen. Enthält die Struktur einen String, der kürzer als die maximale Länge ist, so befinden sich dahinter einige Bytes, die unbenutzt sind und so Datenmüll enthalten können. Hier kann es beim Vergleich zu einem falschen Ergebnis kommen, weil Strukturen Byte-für-Byte - einschließlich des Datenmülls -verglichen werden.
Wenn Sie sicher sind, dass dieses Problem nicht auftreten kann, dann können Sie IgnoreWarning verwenden, um die Warnmeldung zu unterdrücken. IgnoreWarning wirkt nur auf die direkt auf IgnoreWarning folgende Anweisung.

Syntax: IgnoreWarning

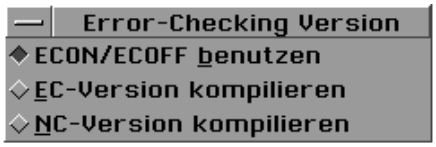
Beispiel

```
DIM t1, t2 AS GeodeToken
  IgnoreWarning
  IF t1=t2 then Print "ungleich"   ' keine Warnmeldung
  IF t1=t2 then Print "ungleich"   ' Jetzt wieder Warnmeldung
```

5.2 Die Error-Checking Version

Sehr häufig kommt es vor, dass man während der Programmentwicklung bestimmte Fehlersituationen abfragen möchte, der entsprechende Code im fertigen Programm aber nur stören würde, weil er das Programm z.B. unnötig verlangsamt. Deswegen bietet R-BASIC die Möglichkeit einzelne Codezeilen nur dann zu compilieren, wenn eine "Error-Checking-Version" (EC-Version) compiliert wird. Alternativ kann man auch Codezeilen nur dann compilieren, wenn eine "Non-Error-Checking-Version" (NC-Version) compiliert wird.

Ob eine EC-Version compiliert wird oder nicht können Sie im Menü "Programm" unter "Error-Checking Version" einstellen.



- ECON/ECOFF benutzen
  Dies ist die Defaulteinstellung. Sie können damit für einzelne Codeabschnitte durch Verwendung der Befehle ECON bzw. ECOFF festlegen, ob eine EC-Version (ECON) oder eine NC-Version (ECOFF) compiliert werden soll.
  Wenn Sie keinen dieser Befehle verwenden wird die NC-Version compiliert, so als ob Sie den Befehl ECOFF geschrieben hätten.
  Hinweis: Das Aktivieren der EC-Version mit ECON muss für jedes Code-Window extra erfolgen!
- EC-Version compilieren
  Es wird in jedem Fall die EC-Version compiliert. Die Anweisungen ECON und ECOFF werden ignoriert.
- NC-Version compilieren
  Es wird in jedem Fall die NC-Version compiliert. Die Anweisungen ECON und ECOFF werden ignoriert.

ECON

ECON aktiviert das Compilieren der EC-Version für die nächsten Codezeilen bis zum nächsten ECOFF-Befehl oder bis zum Ende des Codewindows. Zeilen mit vorangestellter EC Anweisung werden compiliert, Zeilen mit vorangestellter NC Anweisung werden als Kommentarzeilen behandelt.
ECON kann sowohl im BASIC Code als auch im UI Code, auch innerhalb von Objektdeklarationen, verwendet werden.
Damit ECON wirkt muss im Menü "Programm"-"Error-Checking Version" die Option "ECON/ECOFF benutzen" aktiv sein.

Syntax: ECON

Hinweis: Im Modus "ECON/ECOFF benutzen" ist am Beginn jedes Codewindows die NC-Version aktiviert. EC-Code wird erst compiliert, wenn das erste ECON in diesem Codewindow ausgeführt wurde.

ECOFF

ECOFF aktiviert das Compilieren der NC-Version für die nächsten Codezeilen bis zum nächsten ECON-Befehl oder bis zum Ende des Codewindows. Zeilen mit vorangestellter NC Anweisung werden compiliert, Zeilen mit vorangestellter EC Anweisung werden als Kommentarzeilen behandelt.
ECOFF kann sowohl im BASIC Code als auch im UI Code, auch innerhalb von Objektdeklarationen, verwendet werden.
Damit ECOFF wirkt muss im Menü "Programm"-"Error-Checking Version" die Option "ECON/ECOFF benutzen" aktiv sein.

Syntax: ECOFF

EC

EC markiert eine Zeile als "zur EC-Version gehörig". Diese Zeile wird nur compiliert, wenn eine EC-Version compiliert wird, entweder weil die Anweisung ECON ausgeführt wurde oder weil im Menü "Programm"-"Error-Checking Version" die Option "EC-Version compilieren" aktiv ist.

Syntax: EC <Codezeile>

NC

NC markiert eine Zeile als "zur NC-Version gehörig". Diese Zeile wird nur compiliert, wenn eine NC-Version compiliert wird. Das ist in folgenden Fällen der Fall:
- Per Default: Es wurde im aktuellen Codewindow noch keine der Anweisungen ECON oder ECOFF verwendet.
- Die Anweisung ECON wurde durch ein folgendes ECOFF deaktiviert.
- Im Menü "Programm" - "Error-Checking Version" ist die Option "NC-Version compilieren" aktiviert.

Syntax: EC <Codezeile>

Beispiele:

Beispiel 1: Die Sub DrawData funktioniert nur richtig, wenn der Parameter x kleiner als der Parameter y ist. Während der Programmentwicklung wollen wird das überwachen.

```
ECON
SUB DrawData (x, y as Real)
EC IF x >= y THEN MsgBox("Parameterfehler in DrawData")
...
END SUB
```

Beispiel 2: In der EC-Version wollen wir eine Group mit einer Beschriftung versehen, die und darauf aufmerksam macht, das wie eine EC-Version vor uns haben. In der NC-Version soll es nur ein einfacher Text sein.

```
Group InfoKasten
EC  Caption$ ="EC-VERSION Meldungen"
NC  Caption$ ="Meldungen"
...
End Object
```

Beispiel 3: Ein Objekt soll nur in der EC-Version sichtbar sein

```
Button TestButton
    Caption$ ="Test"
NC  visible = FALSE
    ActionHandler = ....
End Object
```

Beispiel 4: Komplexe Verwendung von ECON und ECOFF

```
ECON
    Print "Beispiel"
EC  Print "Error Checking Code 1"
NC  Print "Die Welt ist schön"
ECOFF
EC  Print "Error Checking Code 2"
NC  Print "Der Himmel ist blau"
```

Je nach Einstellung im Menü "Error-Checking Version" werden folgende Texte ausgegeben:

• ECON/ECOFF benutzen

```
Beispiel
Error Checking Code 1
Der Himmel ist blau
```

• EC-Version compilieren

```
Beispiel
Error Checking Code 1
Error Checking Code 2
```

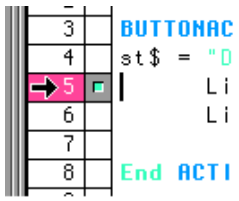
• NC-Version compilieren

```
Beispiel
Die Welt ist schön
Der Himmel ist blau
```

5.3 Der R-BASIC Debugger

Der Debugger ist das leistungsfähigste Werkzeug bei der Fehlersuche. Er ermöglicht es, das Programm an einer bestimmten Stelle anzuhalten, den Programmcode schrittweise auszuführen, die Aufrufreihenfolge der Routinen zu ermitteln, dabei die Werte von Variablen anzusehen und zu ändern und so einen Fehler komfortabel einzugrenzen.

Breakpoints



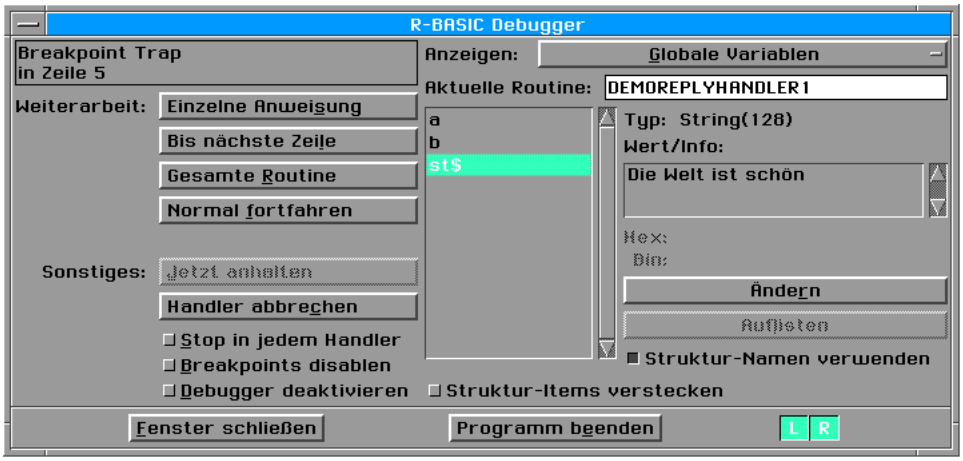
Um ein Programm an einer bestimmten Stelle anzuhalten müssen Sie dort einen Breakpoint (einen Haltepunkt) setzen. Dazu klicken Sie mit der **rechten** Maustaste auf die **Zeilennummer** im Editorfenster.
Sie erkennen einen gesetzten Breakpoint an der rot markierten Zeilennummer. Der nächste rechte Mausklick markiert den Breakpoint als "disabled" (d.h. inaktiv), der dritte löscht ihn wieder.
Breakpoints werden mit dem Programmcode gespeichert und beim Compilieren in das Programm übernommen.
Wenn der Interpreter auf eine Zeile stößt, für die ein Breakpoint gesetzt ist, hält er die Ausführung an und das Debugger-Window öffnet sich. Das Programm befindet sich jetzt im **Einzelschrittbetrieb**. Sie können jetzt das Programm Schritt für Schritt abarbeiten, Variablen einsehen usw.
Hinweise:

- Sie können weitere Breakpoints setzen, Breakpoints disablen oder löschen, während sich das Programm im Einzelschrittbetrieb befindet.
- Breakpoints werden beim Kopieren von Codezeilen **nicht** mitkopiert.
- Um im Einzelschrittbetrieb einen Breakpoint mit der Tastatur zu setzen (Taste F5) müssen Sie vorher im Menü "Programm" die Codefenster editierbar machen.

Laufzeitfehler

Wenn es zu einem Laufzeitfehler kommt, z.B. beim Lesen aus einer nicht geöffneten Datei, wird ein Programm ohne Debugger automatisch beendet. Der Debugger greift hier ein und behandelt einen Laufzeitfehler wie einen Breakpoint. Damit haben Sie Zugriff auf alle Debugger-Funktionen - mit Ausnahme des Einzelschrittbetriebs. Häufig ist jedoch der Schalter "Handler abbrechen" aktiv. Dieser unterbricht den laufenden Handler ohne das Programm zu beenden. In vielen Fällen ist das Programm dann trotz des Laufzeitfehlers weiter lauffähig, was bei der Fehlersuche nützlich sein kann.

5.3.1 Bedienung des Debuggers



Das ist das Debugger-Window. Von hier aus können Sie Ihr Programm steuern, die Variablen einsehen und vieles mehr. Sie können dieses Fenster auch jederzeit über die Taste F10 oder das "Programm" Menü öffnen.

Einzelschrittbetrieb

Steht das Programm in einem Breakpoint so können Sie den Code Zeile für Zeile oder Routine für Routine abarbeiten. Dabei bietet der Debugger folgende Möglichkeiten:

- *Einzelne Anweisung*
  Der Interpreter führt genau eine Anweisung aus und hält dann an der nächsten auszuführenden Anweisung an. Dabei ist es egal, ob die nächste Anweisung in der nächsten Zeile oder irgendwo anders steht. Ist die aktuelle Anweisung z.B. ein Routinenaufruf so wird in die Routine verzweigt und dort angehalten. Wenn die Anweisung eine NEXT-Anweisung ist stoppt der Interpreter entweder in der folgenden Programmzeile oder am zugehörigen FOR-Kommando - je nachdem ob die Schleife beendet ist oder noch ein Schleifendurchlauf folgt.

- *Bis nächste Zeile*
  Der Interpreter setzt einen temporären Breakpoint auf die nächste Zeile und setzt dann die Programmausführung normal fort. Als "nächste Zeile" zählen dabei nur Zeilen, die ausführbaren Code enthalten, als keine Leer- oder Kommentarzeilen. Das Programm stoppt dann am nächsten Breakpoint - also hoffentlich in der nächsten Zeile.
  - Ist die aktuelle Anweisung ein Routinenaufruf, so wird die Routine komplett abgearbeitet und das Programm stoppt, nachdem die Routine zurückkehrt.
  - Ist die aktuelle Anweisung z.B. eine NEXT Anweisung so wird die zugehörige Schleife komplett abgearbeitet. Das Programm stoppt erst, wenn die Schleife verlassen und die auf NEXT folgende Anweisung abgearbeitet wird.
  - Findet der Interpreter vorher einen (anderen) Breakpoint, so stoppt er dort ganz normal. Der temporär gesetzte Breakpoint bleibt dabei erhalten.
  - Der temporäre Breakpoint wird automatisch gelöscht, sobald die zugehörige Zeile ausgeführt wird.
  - Temporäre Breakpoint werden **nicht** in den Sourcecode übernommen.

- *Gesamte Routine*
  Der Interpreter setzt einen temporären Breakpoint auf die Rückkehradresse der aktuellen Routine und setzt dann die Programmausführung normal fort. Die aktuelle Routine wird also komplett abgearbeitet und das Programm stoppt, sobald die Routine zurückkehrt.
  - Findet der Interpreter vorher einen (anderen) Breakpoint, so stoppt er dort ganz normal. Der temporär gesetzte Breakpoint bleibt dabei erhalten.
  - Ist die aktuelle Routine ein Handler, so wird kein Breakpoint gesetzt. Der Handler wird abgearbeitet und das Programm geht in den Standby Modus.

- *Normal fortfahren*
  Die Abarbeitung des Programms wird normal fortgesetzt.

Bereich Sonstiges

- *Jetzt anhalten*
  Mit diesem Schalter weisen Sie den Interpreter an, vor Abarbeitung des nächsten Befehls in den Einzelschrittbetrieb überzugehen. Das ist z.B. hilfreich, wenn das Programm in einer "Endlosschleife" steckt.

- *Handler abbrechen*
  Mit diesem Schalter senden Sie einen "END" Befehl an den Interpreter. Dadurch wird der aktuelle Handler sofort beendet und das Programm geht in den Standby Modus über.

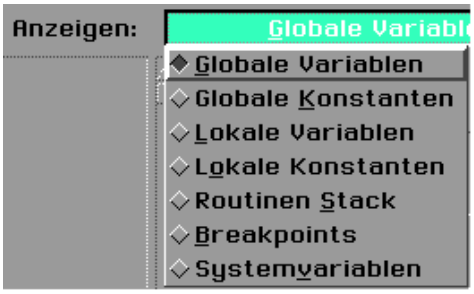
- *Stopp in jedem Handler*
  Wenn diese Option aktiviert ist geht der Interpreter beim Start eines jeden Handlers in den Einzelschrittbetrieb über. Damit kann man z.B. schwer zu findende Fehler eingrenzen, die durch den automatischen Start von Handlern, z.B. dem NotificationHandler eines FileSelectors, verursacht werden.

- *Breakpoints disablen*
  Diese Option bewirkt, dass alle manuell gesetzten Breakpoints ignoriert werden. Dazu zählen aber nicht die temporären Breakpoints der Einzelschrittanweisungen.

- *Debugger deaktivieren*
  Diese Option deaktiviert den Debugger vollständig. Das Programm verhält sich jetzt exakt so, als wenn es als "Eigenständiges Programm" gestartet wäre.

Elemente auf der rechten Seite

- *Aktuelle Routine*
  Dieses Feld enthält den Namen der Routine, in der der Breakpoint oder der Laufzeitfehler aufgetreten sind.



- *Selektor "Anzeigen"*
  Hier wählen Sie aus, welche Elemente in der Liste darunter angezeigt werden sollen. Von dieser Auswahl hängt ab, welche Informationen rechts neben der Liste angezeigt werden.

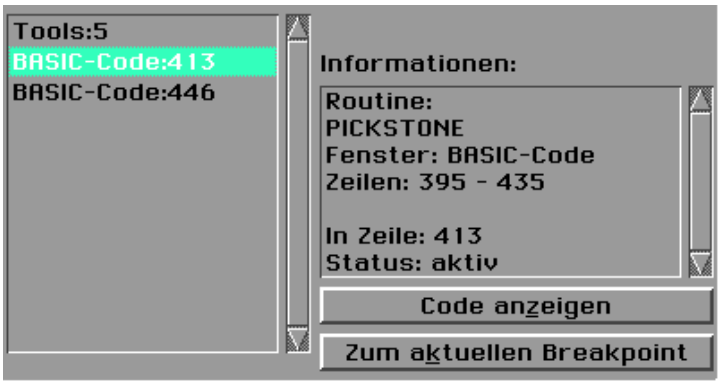
- *Globale und lokale Variablen und Konstanten, Systemvariablen*
  Hier können Sie die Werte von Variablen und Konstanten ansehen. Einfache Variablen und Strukturelemente (Strings und allen numerischen Typen) können verändert werden, Felder und Strukturen können aufgelistet werden. Numerische Variablen werden zusätzlich in ihrer hexadezimalen Darstellung (gerundet, max. 32 Bit) und in ihrer binären Darstellung (gerundet, max. 16 Bit) angegeben.
- Unter "Systemvariablen" können Sie systemweite Variablen wie **fileError**, **currentPath$**, **Screen** und die Systemstrukturen **graphic** (steuert die Grafikausgabe), **printFont** (steuert die Zeichenausgabe mit Print) und **numberFormat** (steuert die Zahlenformatierung) einsehen und teilweise ändern.

  - *Struktur-Items verstecken*
    Diese Option bewirkt, dass in der Liste links die Elemente von Strukturen (erkennbar an einem vorangestellten ~.) nicht angezeigt werden.
  - *Struktur-Namen verwenden*
    Diese Option beeinflusst die Art, wie Strukturen und Arrays von Strukturen vom Debugger aufgelistet werden.

- *Routinen Stack*
  In diesem Bereich wird die Aufrufreihenfolge der Routinen angezeigt. Damit können Sie genau verfolgen, welche Routine von wo aus aufgerufen wurde.

  - *Gehe zu*
    Wechselt zum Code der Routine, die in der Liste links ausgewählt ist.
  - *zu lokalen Variablen wechseln*
    Ist diese Option aktiv so wird mit "Gehe zu" automatisch die Liste der lokalen Variablen der ausgewählten Routine geladen und angezeigt.
  - auch Schleifeneinträge anzeigen
    Jede Schleife (FOR-NEXT, REPEAT-UNTIL, WHILE-WEND) erzeugt einen Eintrag auf dem Stack. Diese Einträge sind hier normalerweise verborgen. Hinweis: Da auf dem Stack die Rückkehradressen in die Routine gespeichert sind erscheinen die zu einer Routine gehörenden Schleifen-Stackeinträge vor dem Namen der Routine (also in der Liste oberhalb des Namens).

- *Breakpoints*
  Hier werden alle im Programm vereinbarten Breakpoints angezeigt. Sie bekommen detaillierte Informationen, wo der Breakpoint vereinbart ist und über seinen Status. Sie können zum Code des ausgewählten Breakpoints oder zu dem Breakpoint wechseln, an dem das Programm angehalten hat.



Elemente in der ReplyBar

Neben den selbst erklärenden Schaltern "Fenster schließen" und "Programm beenden" finden Sie hier Buttons mit den Aufschriften " L " und " R ". Diese Buchstaben stehen für "links" und "rechts" und blenden die entsprechenden Bereiche des Debugger-Windows aus bzw. ein. Damit wird die Größe des Debugger-Windows reduziert, was von Vorteil sein kann, wenn neben dem Debugger-Window gleichzeitig das zu analysierende Programm auf dem Bildschirm zu sehen sein soll. Werden beide Bereiche ausgeblendet wird trotzdem eine minimale UI angezeigt, mit der man das Programm in Einzelschrittbetrieb steuern kann.
Anmerkung: Die IDE selbst kann nicht auf diese Weise versteckt werden. Schieben Sie sie einfach an den unteren Bildschirmrand, wenn sie stört.

5.3.2 Debuggen von Libraries

Die aktuelle Version des Debuggers unterstützt das Debuggen von Code in Libraries nicht. Breakpoints innerhalb von Libraries werden ohne Warnung ignoriert. Um Code aus Libraries zu debuggen müssen Sie ihn ins Hauptprogramm verschieben.

5.3.3 Interne Organisation

Dieser Abschnitt enthält Hintergrundinformationen, deren Kenntnis oder Verständnis für die Arbeit mit dem Debugger nicht unbedingt erforderlich sind.

Wenn der Compiler ein Programm übersetzt, ersetzt er die Variablennamen durch ihre Position im Variablenspeicher, beim Aufruf von Routinen wird statt des Namens die Position der aufgerufenen Routine im Code abgespeichert usw. Wenn der Debugger auf die entsprechenden Namen zugreifen will, müssen sie extra gespeichert werden. Dazu legt der Compiler eine zusätzliche Datei an, die Debugger-Datei genannt wird. Sie wird nur benötigt, wenn das Programm aus der IDE heraus gestartet wird und enthält alle für den Debugger nötigen Informationen, die im eigentlichen Programm nicht benötigt werden. Dazu zählen insbesondere Listen mit den Namen der globalen Variablen, Konstanten und Strukturen sowie aller Routinen und der dazugehörigen lokalen Variablen.

Alle Programme benutzen die gleiche Debugger-Datei, sie heißt "PROGRAM SYMBOL FILE" bzw. für Libraries "LIBRARY SYMBOL FILE". Das hat folgende Konsequenz. Nehmen wir an, Sie compilieren zuerst Programm A. Wenn Sie im Anschluss daran das Programm B compilieren, so gehen die Debugger-Informationen von Programm A verloren. Falls Sie das Programm A danach erneut starten wollen, müssen Sie es erneut compilieren, selbst wenn Sie nichts daran geändert haben.

Jedes Mal, wenn der Interpreter eine Codezeile zum Ausführen lädt, prüft er ab, ob das Programm unter der Kontrolle der IDE läuft. Das geht extrem schnell, weil nur ein einziges Bit abgefragt werden muss. Diese Abfrage verlangsamt ein eigenständiges Programm daher faktisch nicht. Nur wenn das Programm unter der Kontrolle der IDE läuft wird eine Routine gestartet, die prüft, ob ein Breakpoint vorliegt und gegebenenfalls das Programm unterbricht und den Debugger startet. Auch diese Routine fragt einzelne Bits ab, ist also ebenfalls recht schnell. Ein Programm unter der Kontrolle der IDE läuft daher faktisch genauso schnell ab, wie ein eigenständiges Programm.

Intern läuft bei einem Breakpoint der BASIC-Thread (das ist der Prozess-Thread des Launchers) in einer Schleife, die darauf wartet, dass ein bestimmtes Bit zurückgesetzt wird. Der UI-Thread des Launchers kommuniziert währenddessen mit der IDE und nimmt Kommandos, wie z.B. "Einzelschritt ausführen" entgegen. Daraufhin setzt er bestimmte Bits und gibt den BASIC Thread wieder frei. Der BASIC-Thread führt dann z.B. genau einen Befehl aus und landet dann wieder in der Warteschleife.

Während der BASIC-Thread in der Warteschleife ist können Sie auf die Variablen des Programms zugreifen. Auch das wird über den UI-Thread des Launchers abgewickelt, denn nur das BASIC-Programm (und nicht die IDE selbst) kennt die Position der Variablen im Variablenspeicher. Die IDE hingegen kennt über die Debugger-Datei den Namen und den Typ der Variablen.
Analog verhält es sich mit dem Routinen-Stack. Der Launcher kennt die Aufrufreihenfolge und die Position bzw. die Returnadresse der Routinen, die IDE ermittelt daraus mit Hilfe der Debugger-Datei den Namen der Routine.

Bei einem Laufzeitfehler wird die gleiche Routine gerufen, die auch bei einem Breakpoint die Kommunikation mit der IDE abwickelt. Deswegen steht bei einem Laufzeitfehler der Zugriff auf Variablen und Routinenstack genauso zur Verfügung wie in einem Breakpoint. Nur das weitere Ausführen des Programms ist naturgemäß nicht möglich.

(Leerseite)