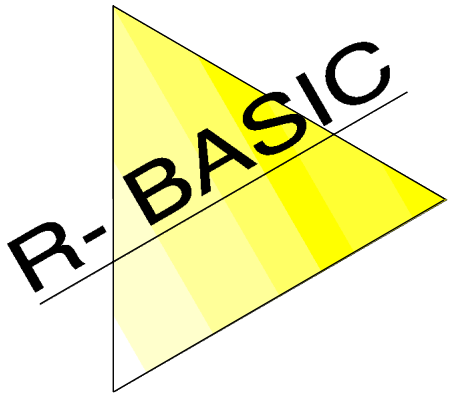
R-BASIC

Einfach unter PC/GEOS programmieren



Programmierhandbuch

Volume 2
Numerische Ausdrücke, Stringausdrücke
Programmablaufsteuerung

Version 1.0

(Leerseite)

Inhaltsverzeichnis

(Leerseite)

2.3 Arbeit mit numerischen Ausdrücken

In diesem Kapitel erfahren Sie alles über die Arbeit mit mathematischen Ausdrücken. Vorher sollten Sie das Kapitel 2.2.2 (Numerische Variablen) gelesen haben.

2.3.1 Grundlagen und Überblick

Die Verarbeitung von Zahlen und mathematischen Funktionen gehört zu den Kernaufgaben einer Programmiersprache. Hier finden Sie eine Zusammenstellung der wesentlichen Dinge, die beim Verarbeiten von Zahlen mit R-BASIC zu beachten sind.

Für Zahlen gelten die folgenden Regeln

- Einfache Zahlen sind z.B. 12 oder 4.89
  Als Dezimaltrenner wird immer der Punkt '.' verwendet, egal was Sie in den PC/GEOS Voreinstellungen festgelegt haben. Dadurch kann man BASIC-Programme auf allen PC/GEOS-Rechnern sofort laufen lassen.
- Vor jede Zahl darf ein Vorzeichen (+ oder –) gesetzt werden.
- Für Zahlen mit 10er-Potenzen wird das E (oder e) verwendet.
  $-3{,}78 \cdot 10^{12}$ wird also so geschrieben: –3.78E12
  $6{,}673 \cdot 10^{-11}$ sieht so aus: 6.673E–11
  Entsprechendes gilt auch für die Ausgabe von Zahlen durch R-BASIC.
- Leerzeichen innerhalb von Zahlen sind unzulässig.
- Zahlen können auch in binärer Schreibweise (Vorsatz &B, z.B. 5 als &B101) und in hexadezimaler Schreibweise (Vorsatz &H, z.B. 243 als &HF3) dargestellt werden. In diesen Fällen sind 32 Bit oder 8 Hexadezimalstellen zulässig (Zahlenbereich DWord). Zahlen in dieser Schreibweise werden grundsätzlich als positive Zahlen behandelt.
- Achtung!
  Bei einer Zahlenbereichsüberschreitung der 1, 2 und 4-Byte Datentypen werden intern die überschüssigen Bits ignoriert. Für die vorzeichenlosen Datentypen (Byte, Word, DWord) entspricht das einer Modulo-Operation.
  Bei vorzeichenbehafteten Werten (Integer und LongInt) führt das dazu, dass aus einer zu großen positiven Zahl eine negative Zahl wird und umgekehrt.
  Beispiel: Die Zuweisung des Wertes 257 zu einer Byte-Variable (Bereich 0 bis 255) führt dazu, dass der Wert 1 gespeichert wird (257 MOD 256 = 1).

Mathematische Operatoren und Vergleiche:

Neben den Grundrechenarten (+, –, *, /) beherrscht R-BASIC die Exponentendarstellung (**^** , z.B. $2^4 = 2\text{^}4$), die Modulo-Operation (MOD, Rest nach Division) sowie die Vergleichsoperatoren =, <, > , <= , >= und <>. Das Ergebnis eines

Vergleichs ist immer wahr (TRUE, siehe unten, numerische Konstanten) oder falsch (FALSE). Eine ausführliche Darstellung der Zusammenhänge finden Sie im Kapitel 2.3.3 (Vergleiche und Operatoren).

Mithilfe von logischen Operatoren (NOT, AND, OR , XOR) und Bit-Operationen (SHL(x, n), SHR(x, n), Shl32(x, n), Shr32(x, n)) können einzelne Bits manipulieren. Mit diesem Thema beschäftigen sich das Kapitel 2.3.5.

Mathematische Funktionen

- R-BASIC verfügt über eine Vielzahl von mathematischen Funktionen. Diese können beliebig verknüpft werden, wobei R-BASIC die üblichen Vorrangregeln (Punktrechnung vor Strichrechnung, Klammern gehen vor usw.) beachtet. Beispiele:

```
y = 4*sin(5*x) + 7
y = sqr( 1 + tan(z))
```

- Überall dort, wo in den Beispielen Zahlen oder numerische Variablen verwendet wurden, können auch komplexe numerische Ausdrücke stehen.
- Zu den mathematischen Funktionen gehören zum Beispiel ABS(x) (absoluter Betrag), INT(x) (ganzzahliger Anteil), SQR(x) (Square root - Quadratwurzel) und SIN(x) (Sinus). Eine vollständige Liste finden Sie im nächsten Abschnitt.

Numerische Konstanten

R-BASIC enthält viele vordefinierte symbolische Konstanten, die anstelle ihrer Zahlenwerte verwendet werden können. In vielen Fällen kann dadurch die Lesbarkeit des Programms verbessert werden. Dazu gehören zum Beispiel Konstanten für die Farbwerte (Farbkonstanten), für Zeichensätze (Fonts) und vieles mehr. Diese Konstanten werden im Zusammenhang mit den entsprechenden Themen besprochen.
Zusätzlich gibt es noch einige "allgemeine" Konstante, die hier aufgeführt sind.

Konstante	Wert	Bedeutung
PI	3,1415...	repräsentiert die Kreiszahl π Beispiel: u = PI*d
FALSE	0	Wahrheitswert "falsch"
TRUE	–1	Wahrheitswert "wahr"
YES	1	Bestätigung
NO	0	Ablehnung

Hinweise:
- Vergleichsausdrücke liefern immer TRUE (–1) oder FALSE (Null) zurück
- R-BASIC behandelt in Entscheidungssituationen (IF...THEN) alle Ausdrücke, die Null ergeben als "falsch", alle Ausdrücke die nicht Null ergeben als "wahr".
- Mit der Anweisung CONST (siehe Kapitel 2.2.10) können Sie sich beliebige Konstanten für eigene Zwecke definieren.

Hierarchie der mathematischen Operatoren

Die mathematischen Operatoren in R-BASIC werden nach einer bestimmten Priorität abgearbeitet. Aus der Schule kennen Sie das als "Punktrechnung geht vor Strichrechnung".

Priorität der Operatoren, hochpriorisierte Operatoren stehen oben:

1. Klammern
2. Exponenten **^**
3. Vorzeichen – , +
4. Multiplikation und Division: *, /, MOD
5. Addition, Subtraktion: +, –
6. Vergleichsoperatoren =, <, >, <=, >=, <>
7. NOT,
8. AND,
9. OR,
10. XOR

Gleichwertige Operatoren werden von links nach rechts abgearbeitet.

Beispiele
Die Prioritäten sichern ab, dass folgende Ausdrücke so abgearbeitet werden, als wären die im Kommentar angegebenen Klammern gesetzt:

```
! Punktrechnung vor Strichrechnung:
  4*A + 7*B           ! --->  (4*A) + (7*B)

! Exponenten-Operator ^ vor den Vorzeichen:
  –5^2                ! --> –(5^2)
  5^–2                ! --> 5^(–2)

! Rechenoperationen vor den Vergleichsoperatoren:
    IF 4•A <= B–1 THEN ..   ---> IF (4•A) <= (B – 1) THEN

! Vergleichsoperatoren vor den logischen Operatoren:
    IF a>4 OR B<2 THEN ..   ---> IF (a>4) OR (B<2 )THEN

! NOT hat die höchste Priorität unter den logischen Operationen
    IF NOT A  AND  NOT B THEN ..
                  ! ---> IF (NOT A)  AND  (NOT B) THEN ..
```

Anmerkung: Ausdrücke mit mehreren verschiedenen logischen Operatoren sind sehr unübersichtlich. Außerdem kommt es sehr schnell zu Fehleinschätzungen der Abarbeitungs-Priorität. Deswegen gibt der Compiler eine Warnung aus, wenn Sie verschiedene logische Operatoren in der gleichen Klammerebene verwenden.

2.3.2 Mathematische Funktionen

Dieses Kapitel beschreibt alle in R-BASIC verfügbaren mathematischen Funktionen. Mathematische Funktionen sind BASIC-Anweisungen, die einen numerischen Wert berechnen. Sie werden in der Form y = ABS(x) verwendet (Ausnahme: RANDOMIZE), wobei y eine numerische Variable ist. Die Grundlagen zu numerischen Variablen finden Sie im Kapitel 2.2.2.

R-BASIC arbeitet mit REAL-Zahlen im Bereich von $\pm 3.9999 \cdot 10^{4931}$. Einige der mathematischen Funktionen haben einen eingeschränkten Definitionsbereich, d.h. sie sind für Argumente (x-Werte) außerhalb eines bestimmten Zahlenbereichs nicht anwendbar. Übergibt man einen x-Wert außerhalb des Definitionsbereichs so liefern die Funktionen einen spezielle "Fehlerwert", es kommt **nicht** zum Programmabbruch! Eine Auflistung der Definitionsbereiche und Fehlerwerte finden Sie im Anhang.

Rechnet man mit den Fehlerwerten (Fehler, Unterlauf und Überlauf) weiter, so bleibt der Fehlerwert erhalten. Gibt man einen Fehlerwert aus (Z.B. PRINT oder Str$(x)), so wird der entsprechende Text ("Fehler", "Überlauf", "Unterlauf" ausgeben.

Beispiel:

```
PRINT SQR(-3)      ! Das Wort "Fehler" erscheint
```

Einfache mathematische Funktionen:

Funktion	Bedeutung
ABS(x)	Absoluter Betrag von x: lxl
SGN(x)	Signum-Funktion (Vorzeichen-Funktion) Liefert –1 (negativ), 0 oder +1 (positiv)
INT(x)	Liefert die nächst kleinere ganze Zahl, d.h. es wird nach unten gerundet: INT(x) ≤ x
TRUNC(x)	Kürzt x auf seinen ganzzahligen Anteil, d.h. es wird Richtung Null gerundet. TRUNC(x) und INT(x) unterschieden sich bei negativen x.
FRAC(x)	Liefert den gebrochenen Anteil von x, d.h. die Nachkommastellen. Das Ergebnis ist immer positiv.
ROUND(x [, n])	Rundet x auf n Stellen nach dem Komma. Wird n weggelassen, erfolgt die Rundung auf ganze Zahlen, (so als ob n = 0 wäre).

Beispieltabelle für die einfachen mathematischen Funktionen:

x	SGN(x)	INT(x)	TRUNC(x)	FRAC(x)	ROUND(x)
3.8	+1	3	3	0.8	4
3.5	+1	3	3	0.5	4
3.1	+1	3	3	0.1	3
0	0	0	0	0	0
- 3.2	- 1	- 4	- 3	0.2	- 3
- 3.5	- 1	- 4	- 3	0.5	- 3
- 3.7	- 1	- 4	- 3	0.7	- 4

Anmerkungen:

- ROUND verwendet bei x.5 "Gerade-Zahl-Regel". Das bedeutet, dass auf die nächste gerade Zahl gerundet wird, auch bei negativen Zahlen. Beispiele:

```
ROUND(3.5) --> 4
ROUND(2.5) --> 2
ROUND(1.5) --> 2
ROUND(–2.5) --> –2
```

- INT(x), TRUNC(x) und FRAC(x) führen vorher keine Rundung aus, sondern nehmen der Wert, wie er intern vorhanden ist. Beim Ausgeben eine Zahl (Print oder Str$) wird jedoch gerundet. Das kann zu scheinbaren Widersprüchen führen:

```
PRINT    4.999999999999          --> 5
PRINT    INT(4.999999999999)     --> 4
PRINT    FRAC(4.999999999999)    --> 1
```

  Sollte das ein Problem sein, runden Sie den Wert vorher.

Zufalls-Zahlen:

Funktion	Bedeutung
RANDOMIZE [n]	Initialisiert den Zufallsgenerator. n: Initialisierungswert (n sollte eine große Zahl sein). Bei gleichem Initialisierungswert liefert der Zufallsgenerator immer die gleiche "zufällige" Folge. Beispiel: RANDOMIZE 1123581321 Ohne n: Der Initialisierungswert wird aus der Systemzeit ermittelt. Der Zufallsgenerator liefert immer verschiedene "zufällige" Folgen.
RND()	Liefert eine Zufallszahl im Bereich 0 <= x < 1

Tipps:

- Initialisieren Sie den Zufallsgenerator nur einmal im Programm, am besten am Programmanfang:

```
RANDOMIZE
```

- Ganzzahlige Zufallszahlen im Bereich von 0...n-1 (jeweils einschließlich) erhält man mit:

```
   y = INT( n * RND())
```

- **Warnung!** RANDOMIZE und RND() verwenden den GEOS-internen Zufallsgenerator. Der Programmierer von R-BASIC kann daher nicht garantieren, dass bei verschieden Systemversionen die RND() -Funktion bei gleichem Initialisierungswert von RANDOMIZE wirklich die gleiche Zufallszahlenfolge liefert. Diese Warnung ist z.B. für kryptografische Programme bedeutsam. Der Programmierer übernimmt diesbezüglich keinerlei Haftung!

INCR und DECR

Die Anweisungen INCR (engl. increment, Zuwachs, Vergrößerung) und DECR (engl. decrement, Verringerung) vergrößern oder verkleinern den Wert einer numerischen Variablen um 1 oder um einen vorgegeben Wert n. N muss ganzzahlig sein, im Bereich von -32768 bis +32767 liegen und zur Compilezeit berechenbar sein. Einfache Berechnungen (+, -, *, /, Klammern, ^ (Exponent) , MOD (Modulo-Division), OR, AND, NOT, XOR und die Funktionen INT(), ASC(), SQR(), FRAC(), TRUC(), SIN(), COS(), TAN(), EXP(), LN(), LOG(), LG() und SizeOf(). sowie negative Werte sind zugelassen. Variablen und sonstige Funktionen sind nicht erlaubt.

Funktion	Bedeutung
INCR <numVar>	Vergrößerung des Variablenwertes um 1
INCR <numVar> , n	Vergrößerung des Variablenwertes um n (n: ganzzahlig)
DECR <numVar>	Verkleinerung des Variablenwertes um 1
DECR <numVar>, n	Verkleinerung des Variablenwertes um n (n: ganzzahlig)

Beispiele:

```
DIM x as REAL
DIM w as WORD
CONST   D_X = 18

  INCR x              ' entspricht x = x + 1
  DECR w, 12          ' entspricht w = w – 12
  INCR w, D_X + 2     ' entspricht w = w + 20
```

INCR und DECR laufen deutlich schneller als ihre Entsprechungen x = x + n bzw. x = x - n.

Transzendente Funktionen:

Funktion	Bedeutung
SQR(x)	Quadratwurzel aus x
EXP(x)	Exponentialfunktion eX
LN(x)	Natürlicher Logarithmus von x, d.h. Logarithmus zur Basis e
LOG(x)	Dekadischer Logarithmus von x, d.h. Logarithmus zur Basis 10, log10(x)
LG(x)	Logarithmus zur Basis 2, log2(x)

Trigonometrische Funktionen:

Funktion	Bedeutung
SIN(x)	Sinus von x, x im Bogenmaß
COS(x)	Cosinus von x, x im Bogenmaß
TAN(x)	Tangens von x, x im Bogenmaß
ASN(x)	ArcusSinus von x - Umkehroperation zu Sinus
ACS(x)	ArcusCosinus von x - Umkehroperation zu Cosinus
ATN(x)	ArcusTanges von x - Umkehroperation zu Tangens

Im Bogenmaß hat ein Vollkreis nicht 360°, sondern den Wert 2π. Die Umrechnungsformel lautet:

```
wert_im_bogenmaß = wert_im_gradmaß * PI / 180
```

Wenn Sie das Argument im Gradmaß haben und mit einer Genauigkeit von 4 Stellen nach dem Komma auskommen können Sie statt der in der Tabelle angegeben Funktionen auch eine der WWFixed-Funktionen FixSin, FixCos, FixTan und FixAsn verwenden. Diese Funktionen können direkt (d.h. ohne Konvertierungsfunktion) in Real-Ausdrücken verwendet werden und erwarten das Argument im Gradmaß. Details zu den WWFixed-Funktionen finden Sie im Kapitel 2.3.6.

Hyperbolische Funktionen:

Funktion	Bedeutung
SinH(x)	Sinus Hyperbolicus von x
CosH(x)	Cosinus Hyperbolicus von x
TanH(x)	Tangens Hyperbolicus von x
ASNH(x)	Arcus Sinus Hyperbolicus von x
ACSH(x)	Arcus Cosinus Hyperbolicus von x
ATNH(x)	Arcus Tangens Hyperbolicus von x

2.3.3 Operatoren und Vergleiche

R-BASIC verfügt über die im Folgenden angegebenen mathematischen Operatoren. Eine Übersicht über die Hierarchie (Abarbeitungsreihenfolge) der Operatoren finden Sie vorn, im Kapitel 2.3.1 (Grundlagen)

Einfache Operatoren:

Operator	Funktion
+, –	Addition und Subtraktion
*, /	Multiplikation und Division.
^	Exponent. z.B. 2^3 entspricht 2^3
MOD	Modulo-Operation: Division mit Rest, wobei der Rest das Ergebnis ist. Beispiele: 7 MOD 3 liefert 1 denn 7/3 = 2 Rest 1 3.4 MOD 1.3 liefert 0.8 denn 3.4 = 2•1.3 + 0.8

Tipp: Die Zeichen ^ (Exponent) * (Sternchen für Multiplikation) und - (Minus) sind eventuell nicht oder nur schwer am Bildschirm zu identifizieren. R-BASIC unterstützt daher für diese Zeichen Ersatz-Zeichen, die Sie stattdessen schreiben können, um die Lesbarkeit Ihres Codes zu verbessern.

^ : Exponent-Ersatzzeichen : ** (zwei Sternchen)
* : Multiplikation-Ersatzzeichen : • (AltGr+Shift+8, ASCII-Code 165)
- : Minus-Ersatzzeichen : – (AltGr+Minus, ASCII-Code 208)

Vergleichsoperatoren

Vergleichsoperationen liefern Wahr (TRUE, –1) oder Falsch (FALSE, 0)

Operator	Syntax	Funktion
=	A = B	Wahr, wenn A gleich B ist
<	A < B	Wahr, wenn A kleiner als B ist
>	A > B	Wahr, wenn A größer als B ist
<=	A <= B	Wahr, wenn A kleiner oder gleich B ist
>=	A >= B	Wahr, wenn A größer oder gleich B ist
<>	A <> B	Wahr, wenn A ungleich B ist

Die Vergleichsoperatoren (<, <=, >, >=, = <>) stehen auch für Zeichenketten zur Verfügung. Für Variablen vom Typ FILE, HANDLE oder OBJECT sowie für Strukturen stehen die Vergleichsoperatoren = und <> zur Verfügung.
Hinweis: Statt A = B kann man für Vergleich auch ein doppeltes Gleichheitszeichen schreiben (A == B). Das verbessert gelegentlich die Lesbarkeit.

Logische Operatoren

Logische Operatoren wirken bitweise auf die Operanden. Eine ausführliche Erklärung sowie Beispiele finden Sie im nächsten Kapitel.

Operator	Syntax	Funktion
NOT	NOT A	Negation Liefert immer das Gegenteil
AND	A AND B	Logisches UND Liefert Wahr, wenn beide Werte wahr sind.
OR	A OR B	Logisches ODER Liefert Wahr, wenn mindestens einer der Werte wahr ist.
XOR	A XOR B	Logisches Exklusiv ODER Liefert Wahr, wenn entweder der eine oder der andere Wert wahr ist. Sind beide Werte Wahr, liefert XOR Falsch.

Beispiele:
Da Vergleiche höher priorisiert sind benötigt man hier keine Klammern.

```
NOT 4 > 7              ' liefert Wahr
3 > 7 OR 5 > –2        ' liefert Wahr
3 > 7 AND 5 > –2       ' liefert Falsch
3 > 7 XOR 5 > –2       ' liefert Wahr
```

2.3.4 Bits, Bytes, Binär- und Hexadezimalzahlen

Als fortgeschrittener Programmierer kommt man letztlich um binär und Hexadezimaldarstellungen nicht herum. Als Anfänger sollte man dieses Kapitel zumindest überfliegen, damit man eine Vorstellung davon bekommt, was das Ganze eigentlich soll.

Um zu verstehen, wie Computer Zahlen darstellen, müssen wir uns zunächst darüber klar werden, wie wir das eigentlich selber im täglichen Umgang mit Zahlen machen.

Unser "normales" Dezimalsystem hat 10 Ziffern: 0 bis 9 - damit könnten wir genau 10 Zahlen darstellen, nämlich 0 bis 9. Wenn wir größere Zahlen darstellen wollen, setzen wir einfach weitere Ziffern davor - in 47 bedeutet die 4 eigentlich 4•10 und in 398 bedeutet die 3 in Wirklichkeit 3•100. Das hat man so vereinbart und jeder hält sich daran.
Da $100 = 10^2$ ist, $10 = 10^1$ und $1 = 10^0$, kann man sagen, dass gilt:

$$398 = 3 \cdot 10^2 + 9 \cdot 10^1 + 8 \cdot 10^0$$

Dieses Prinzip wenden wir auf alle Zahlen an, wobei die 10er-Potenzen daher kommen, dass wir eben 10 Ziffern (0 bis 9) haben. Man beachte, dass, obwohl wir 10 Ziffern haben, bereits die Zahl 10 zweistellig ist - weil wir ja mit Null beginnen.

Computer kennen nur zwei Ziffern: 0 und 1 (entsprechend Strom an und Strom aus). Sie sind also gezwungen bereits eine Ziffer "davor" zu setzen, wenn sie die Zahl Zwei darstellen wollen. Die Zahlendarstellung mit nur zwei Ziffern nennen wir "binär" (bzw. Binärsystem).

Analog zu unserer üblichen Vereinbarung, dass

$$40 = 4 \cdot 10^1 + 0 \cdot 10^0$$

ist, gilt für Binärzahlen ebenfalls

$$10 = 1 \cdot 2^1 + 0 \cdot 2^0$$

bzw. $11 = 1 \cdot 2^1 + 1 \cdot 2^0$

Wir verwenden Zweier-Potenzen, das wir genau zwei Ziffern (0 und 1) haben.
Dieses Prinzip kann man auf beliebig lange Binärzahlen anwenden. Eine weitere Vereinbarung aus unserem üblichen Dezimalsystem, nämlich dass führende Nullen zulässig sind (es ist egal ob wir 7 oder 007 schreiben), ermöglicht uns, die folgende Tabelle aufzustellen:

binär	Zweierpotenz	dezimal
0000	keine	0
0001	2^0	1
0010	2^1	2
0100	2^2	4
1000	2^3	8

Jede Zahl kann man als Summe dieser "elementaren" Zweierpotenzen darstellen, z.B. ist

$$1010 = 2^3 + 2^1 = 8 + 2 = 10$$

In der Computertechnik wird eine einzelne Binärstelle als **Bit** bezeichnet. Ist der Wert 1, sagt man, das Bit ist "gesetzt", andernfalls ist es "nicht gesetzt". Es ist üblich, die Bits von rechts beginnend durchzunummerieren, wobei die ganz rechte Stelle als **Bit Null** bezeichnet wird. Das hat für den Mathematiker den Vorteil, dass die Bitposition gleich dem Potenzwert ist (Bit 0 : 2^0, Bit 1 : 2^1 usw.).
Im Beispiel oben (1010) sind also die Bits 1 und 3 gesetzt.

Mit einem Bit kann man $2^1 = 2$ Zahlen darstellen, bei zwei Bit sind es bereits $2^2 = 4$ Zahlen (00, 01, 10 und 11) und mit 4 Bit sind es $2^4 = 16$ Zahlen. Das ist noch nicht sehr viel. Daher fasst man 8 Bit zu einem **Byte** zusammen. Ein Byte enthält also die Bits 0 bis 7:

7	6	5	4	3	2	1	0

Damit kann man $2^8 = 256$ verschiedene Zahlen darstellen (0 bis 255). Wenn das nicht reicht, nimmt man 16 Bit, ein sogenanntes **Word**. Hier kommt man auf $2^{16} = 65536$ Zahlen (0 bis 65535). Für gehobene Ansprüche gibt es noch das **DWord** (Double word) mit 32 Bit, dort reicht der Zahlenbereich bis 4294967295.

Bereits bei einem Byte wird die Binärdarstellung unübersichtlich. Man erfasst den Unterschied zwischen 10010101 (= 149) und 10101001 (=169) nicht mehr auf den ersten Blick. Deswegen hat es sich als praktisch erwiesen, jeweils 4 Binärziffern zusammenzufassen. Mit vier Binärziffern (4 Bit) kann man aber 16 Zahlendarstellen - die Ziffern 0 bis 9 reichen da nicht mehr. Man behilft sich daher mit den ersten Buchstaben des Alphabets (konkret A bis F). Diese Darstellung nennt man **hexadezimal** (hexa = 6, dezi = 10).

binär	hexadezimal	dezimal
0000	0	0
0001	**1**	**1**
0010	**2**	**2**
0011	3	3
0100	**4**	**4**
0101	5	5
0110	6	6
0111	7	7
1000	**8**	**8**
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

Nach einiger Übung kommt man mit dieser zunächst sehr exotisch anmutenden Zahlendarstellung gut zurecht und wird bald die Vorteile zu schätzen wissen
Einige Beispiele:

1001 0101 = 95 (hex.)	(= 149 dez.)
1010 1001 = A9 (hex.)	(= 169 dez.)
1111 1111 = FF (hex.)	(= 255 dez.)
1100 0111 = C7 (hex.)	(= 199 dez.)

Aus der Tabelle wird ein Problem ersichtlich: ohne Kommentar kann man nicht entscheiden, ob z.B. 1001 dezimal oder binär gemeint ist (oder sogar hexadezimal?). Daher ist es in Programmiersprachen übliche, das Zahlensystem zu kennzeichnen. In R-BASIC gilt folgende Vereinbarung:

ohne Kennung: dezimal	z.B. 12
Kennung &H: hexadezimal	z.B. &H95 (= 149 dez.)
Für C-Programmierer: 0x: Hexadezimal	z.B. 0x95 (= &H95 = 149 dez.)
Kennung &B: binär	z.B. &B1100 (= 12 dez.)

Rein formal kann man auch beim Hexadezimalsystem mit der Potenzdarstellung arbeiten, nur dass hier die Basis 16 verwendet werden muss (beachte: $16^0 = 1$):

$$\&h95 = 9 \cdot 16^1 + 5 \bullet 16^0 = 144 + 5 = 149$$

$$\&hA9 = A \cdot 16^1 + 9 \bullet 16^0 = 10 \cdot 16^1 + 9 \bullet 16^0 = 160 + 9 = 169$$

Dieses Verfahren kann man verwenden, wenn man Hexadezimalzahlen "von Hand" in Dezimalzahlen umwandeln muss. Die meisten wissenschaftlichen Taschenrechner verfügen aber heute über entsprechende Funktionen.

R-BASIC bietet über das Menü "Extras" -> "Tools" ein kleines Programm an (**HBDConverter**, © by John Howard and used by permission), mit dem man Binär-, Dezimal- und Hexadezimalzahlen ineinander umrechnen kann. Das Programm ist in der Standardinstallation von R-BASIC nicht enthalten, es muss separat von der R-BASIC-Webseite heruntergeladen werden.

Außerdem gibt es die Stringfunktionen **Hex$( )** und **Bin$( )**, mit denen Sie Zahlen in hexadezimaler und binärer Darstellung ausgeben können.

Was passiert eigentlich, wenn man bei einem Byte zu der größten darzustellenden Zahl (255 = &HFF) noch Eins addiert? Eigentlich kommt ja 256 (=&H100) heraus. Im Binärsystemsieht das so aus:

```
    1111 1111
+          1
-----------
1 0000 0000
```

Diese Zahl hat 9 Bit und kann in einem Byte nicht mehr dargestellt werden. Man hat festgelegt, dass in einem solchen Fall, die wir **Überlauf** nennen, die führenden Bits, die nicht in das Byte passen, ignoriert werden: 255 + 1 ist also in diesem Fall Null. "In diesem Fall" heißt, dass wir das Ergebnis in einem Byte speichern wollen. Haben wir ein Word (16 Bit) zur Verfügung, können wir das Ergebnis sehr wohl

abspeichern und es kommt 256 heraus. Erst 65535 + 1 ergäbe 65536, was nicht mehr in ein Word passt und daher wieder Null ergibt. Dieses Phänomen muss man kennen, wenn man mit Bits und Bytes direkt arbeitet.

Das Problem des Überlaufs führt immer wieder zu schwer auffindbaren Fehlern. In R-BASIC sollte daher nach Möglichkeit der Datentyp REAL (Genauigkeit 10 Byte) verwendet werden. Da intern alle Berechnungen mit REAL-Zahlen ausgeführt werden, ist dieser Datentyp auch fast am schnellsten. Nur WWFixed ist schneller.

Wenn Sie die "kleinen" Datentypen verwenden müssen, stehen Ihnen in R-BASIC die Typen Byte, Word , Integer, DWord, LongInt (4 Byte, mit Vorzeichen) sowie der Typ WWFixed zur Verfügung. Im Kapitel 2.2.2 (Numerische Datentypen und numerische Ausdrücke) finden Sie weitere Informationen dazu.

Und nun noch die verrückten Mathematiker ... (oder: Futter für Fortgeschrittene)

Ein Mathematiker wird sich schnell beschweren, dass es keine negativen Zahlen gibt. Aber ihm kann geholfen werden. Wie bereits mehrfach erwähnt, basiert bei der Zahlendarstellung sehr viel auf Vereinbarungen.
Wie wir gerade gesehen haben ergibt, wenn wir ein Byte binär betrachten,

1111 1111 + 1 = 0

Also muss gelten:

0 – 1 = 1111 1111

0–1 ist aber –1. Es ergibt daher Sinn, wenn man negative Zahlen benötigt, festzulegen, dass alle Zahlen, deren führendes Bit gesetzt ist (beim Byte also Bit 7, beim Word das Bit 15), negative Zahlen sein sollen.

0000 0000 = 0
1111 1111 = -1
1111 1110 = -2
usw. bis
1000 0000 = - 128

Man kann in einem Byte also Zahlen von –128 bis +127 darstellen - insgesamt wieder 256 verschiedene Zahlen. Man nennt diese Darstellung "Zweierkomplement".
Offensichtlich ist es nur eine Frage der Vereinbarung, ob man die Binärzahl 1111 1111 als -1 oder als + 255 auffasst. Das ist sicher anfänglich sehr verwirrend und zum Glück benötigt man diese Informationen nur sehr selten.
Dem Computer ist das letztlich egal, denn die Rechenregeln sind so aufgestellt, dass er immer so tun kann, als seien alle Zahlen positiv. Tun wir es ihm nach.

2.3.5 Logische Operatoren und Bit-Operationen

In R-BASIC sind die folgenden bitweisen logischen Operatoren und Funktionen definiert:

Operator / Funktion	Funktion
NOT A	bitweise Negation
A AND B	bitweises logisches UND
A OR B	bitweises logisches ODER
A XOR B	bitweises logisches Exklusiv ODER
SHL	bitweises Linksschieben, 16 Bit
Shl32	bitweises Linksschieben, 32 Bit
SHR	bitweises Rechtsschieben, 16 Bit
Shr32	bitweises Rechtsschieben, 32 Bit

Zur Arbeit mit den bitweisen logischen Operationen sind die folgenden Konstanten hilfreich:

Konstante	Wert	Bedeutung
FALSE	0	Wahrheitswert "falsch"
TRUE	–1	Wahrheitswert "wahr"

TRUE ist als -1 definiert, weil in der Binärdarstellung dann alle Bits gesetzt sind. Das ermöglich eine einfache Zusammenarbeit von Vergleichen (sie liefern TRUE oder FALSE) und logischen Operationen (z.B. AND oder OR).

2.3.5.1 Logische Ausdrücke in Entscheidungen

In einigen Situationen (z.B. bei den Anweisungen IF und WHILE, siehe Kapitel 2.5) muss R-BASIC Entscheidungen treffen. Dazu wird ein numerischer Ausdruck ausgewertet. In vielen Fällen ist das einfache Vergleichsoperation. Beispiele:

```
IF A > B THEN ...
IF (A + B) > 12 THEN ...
WHILE A > 0
  ...
  WEND
```

Gelegentlich müssen aber mehrere Bedingungen erfüllt sein. Zum Beispiel kann es sein, dass A>B **und** C<0 gleichzeitig gelten muss. Oder es reicht aus wenn eine der Bedingungen X>Y **oder** Z<0 erfüllt ist.
Natürlich kann man die Bedingungen nacheinander abfragen. Aber eleganter und effizienter ist es, die Bedingungen durch logische Operationen zu verknüpfen.

Wenn R-BASIC eine Vergleichsoperation ausführt kann das Ergebnis wahr (TRUE) oder falsch (FALSE) sein. Für die Verknüpfung mithilfe logischer Operatoren gilt folgendes:

- **NOT** A ist wahr, wenn A falsch ist und umgekehrt.
- A **AND** B ist wahr, wenn **sowohl** A **als auch** B wahr sind
- A **OR** B (logsiches ODER) ist wahr, wenn A **oder** B, **oder beide** wahr sind
- A **XOR** B (exklusives ODER) ist wahr, wenn **entweder** A **oder** B wahr sind. Sind beide wahr liefert A XOR B falsch.

Die folgende Tabelle verdeutlicht das. A und B seien Ergebnisse einer Vergleichsoperation, die wahr (**W**, TRUE, –1) oder falsch (**F**, FALSE, 0) sein können.

A	B	NOT A	A AND B	A OR B	A XOR B
F	F	W	F	F	F
F	W	W	F	W	W
W	F	F	F	W	W
W	W	F	W	W	F

Beispiele
Da die Vergleichsoperatoren höher priorisiert sind als die logischen Operatoren führt R-BASIC zuerst die Vergleiche aus und verknüpft deren Ergebnisse anschließend mit den logischen Operationen. Wenn Sie mehrere logische Operatoren verwenden sollten Sie Klammern setzten.

```
IF  A > 0  THEN ...
IF  A > 0  OR   Name$ = "Paul"  THEN ...
IF ( A >= 0  AND  A <= 9 ) OR ( A >= 100 AND  A <= 109 ) THEN ...

REPEAT
  ....
  UNTIL C > 0 AND D = 7
```

Wenn R-BASIC eine Entscheidung trifft (IF, WHILE, UNTIL) prüft es, ob der entsprechende numerische Ausdruck Null ist oder nicht. Jeder Wert, der **nicht Null** ist, wird als **wahr** angesehen. Folgende Formulierungen sind daher gleichwertig:

```
  IF A <> 0 THEN ...
  IF A THEN ...
```

Ein Wert kann also "wahr" sein (z.B. 12) ohne TRUE (–1) zu sein. Vermeiden Sie daher Formulierungen wie die folgende, das kann zu schwer zu findenden Fehlern führen.

```
' Möglicherweise fehlerhafter Code:
  IF   A = TRUE   THEN ..      ' ist nur erfüllt, wenn A den
                               ' Wert –1 hat
```

2.3.5.2 Anwendung der logischen Operatoren auf Zahlen

Achtung! Das ist ein sehr komplexes, aber auch ein sehr leistungsfähiges Thema. Kenntnisse in Logik, der Binär- und Hexadezimal-Darstellung von Zahlen sind hilfreich.

Intern werden die logischen Operatoren bitweise auf 16-Bit Zahlen (Datentyp Word) angewandt. Andere Datentypen werden vorher in den Datentyp Word konvertiert.
Dadurch können die logischen Operatoren auch auf alle Zahlen angewendet und in numerischen Ausdrücken verwendet werden. Beispiele:

```
DIM A, B, Y
  A = 3          ' einfache Zuweisung
  B = 7 AND A
  Y = B  OR  4
  Y = NOT ( ( A + 5 )  AND  7 )
```

Wichtig ist das Setzen von Klammern. Der Compiler hat zwar eine bestimmte Hierarchie bei der Abarbeitung der logischen Operatoren, es kommt hier jedoch sehr schnell zu Fehleinschätzungen von Seiten des Programmierers.
Insbesondere gilt, dass auch hinter NOT stets der gesamte numerische Ausdruck ausgewertet wird. Klammern beschränken den Wirkungsbereich von NOT nicht, weil NOT keine Funktion sondern ein Operator ist. Die Anweisung

```
y = NOT(4) + 1
```

ist deshalb identisch mit

```
y = NOT 5    ' bzw y = NOT ( (4) + 1 )
```

und **nicht** mit

```
y = ( NOT 4 ) + 1
```

Zum Verständnis der Wirkung von logischen Operatoren auf Zahlen muss man die Zahlen in die Binär-Darstellung umwandeln. Binärzahlen sind im Kapitel 2.3.4 erklärt.
Bei der logischen Verknüpfung von Zahlen passiert folgendes:

- Die Operanden werden in ein 16-Bit-Bitmuster umgewandelt (Datentyp: Word)
- Die Bitmuster werden entsprechend der folgenden Tabelle bitweise miteinander verknüpft.

Tabelle: Logische Operatoren bei Anwendung auf einzelne Bits

A	B	NOT A	A AND B	A OR B	A XOR B
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Beispiel für die Anwendung logischer Operatoren auf Zahlen. Aus Platzgründen werden in der Binärdarstellung nur die unteren 4 Bit betrachtet. In Klammern steht die zugehörige Dezimalzahl.

A	B	A AND B	A OR B	A XOR B
0011 (= 3)	0100 (= 4)	0000 (= 0)	0111 (= 7)	0111 (= 7)
0100 (= 4)	0111 (= 7)	0100 (= 4)	0111 (= 7)	0011 (= 3)
0111 (= 7)	0011 (= 3)	0011 (= 3)	0111 (= 7)	0100 (= 4)

Die logische Operation NOT negiert alle Bits. Die folgende Tabelle verdeutlicht das.

A	Binärdarstellung	NOT A (binär)	NOT A
3	0000 0000 0000 0011	1111 1111 1111 1100	65532
7	0000 0000 0000 0111	1111 1111 1111 1000	65528
TRUE (–1)	1111 1111 1111 1111	0000 0000 0000 0000	0 (FALSE)

Wenn Sie mit Hilfe der logischen Operatoren Bit-Manipulationen vornehmen, sollten Sie Variablen vom Datentyp WORD oder BYTE verwenden, um unerwartete Ergebnisse zu vermeiden. Insbesondere kann es passieren, dass negative Zahlen auftreten. Dies ist kein Fehler, denn es ist auch möglich, negative Zahlen mit logischen Operatoren zu verknüpfen. Dazu muss man folgendes wissen:

- Logische Operatoren arbeiten gleichwertig mit vorzeichenbehafteten und mit vorzeichenlosen 16-Bit-Zahlen.
- Eine vorzeichenbehaftete 16-Bit-Zahl ist negativ, wenn das höchstwertige Bit (Bit 15) gesetzt ist.
- Die Zusammenarbeit von logischen Operatoren mit den Vergleichsoperatoren funktioniert deshalb, weil in – 1 alle Bits gesetzt sind, während in der Null kein Bit gesetzt ist. Deswegen ist TRUE als – 1 definiert
- Die bitweise Verwendung von vorzeichenbehafteten Ganzzahlen macht selbst erfahrenen Programmieren oft Mühe.

Mann kann die logischen Verknüpfungen von Zahlen auch direkt in einer IF-Anweisung (bzw. WHILE-WEND oder REPEAT-UNTIL-Anweisung) verwenden. Allerdings muss man hier beachten, dass jede Zahl, die nicht Null ist, als "wahr" interpretiert wird. Im folgenden Codefragment wird deshalb der THEN-Zweig **nicht** abgearbeitet. Der Grund ist, das die logische Verknüpfung 4 AND 2 den Wert Null liefert.

```
DIM A, B

  A = 4
  B = 2
  IF A AND B THEN
    ....
  End IF
```

2.3.5.3 Bit-Schiebe-Operationen

Um die in diesem Abschnitt beschriebenen Kommandos verwenden zu können sollten Sie sich mit der Binärdarstellung von Zahlen (siehe Kapitel 2.3.4) auskennen.

SHL, Shl32

Die Funktion SHL (Shift Left - schiebe nach links) führt eine bitweise Schiebeoperation auf ein 16-Bit-Word aus. Shl32 führt diese Operation auf ein 32-Bit-DWord aus. Die niederwertigen Bits werden mit Null aufgefüllt, die höchstwertigsten Bits gehen verloren.

Syntax: **<numVar>** = **SHL** (**x**, **n**)
<numVar> = **Shl32** (**x**, **n**)
x:numerischer Ausdruck
16-Bit-Word bei SHL, 32-Bit-DWord bei Shl32
n: Anzahl der Bits, um die geschoben werden soll.

Bit-Nr: 14 12 10 8 6 4 2 0

Beispiel:

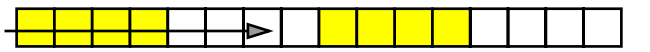
```
y = SHL(7, 2)
' 7 ist binär 000111, also  ist y = 011100 (binär), d.h. y = 21
```

SHR, Shr32

Die Funktion SHR (Shift Right - schiebe nach rechts) führt eine bitweise Schiebeoperation auf ein 16-Bit-Word aus. Shr32 führt diese Operation auf ein 32-Bit-DWord aus. Die höchstwertigen Bits werden mit Null aufgefüllt, die niederwertigsten Bits gehen verloren.

Syntax: **<numVar>** = **SHR** (**x**, **n**)
<numVar> = **Shr32** (**x**, **n**)
x:numerischer Ausdruck
16-Bit-Word bei SHL, 32-Bit-DWord bei Shl32
n: Anzahl der Bits, um die geschoben werden soll.

Bit-Nr: 14 12 10 8 6 4 2 0



Beispiel:

```
y = SHR(12, 2)
' 12 ist binär 001100, also ist y = 000011 binär, d.h. y =
```

2.3.5.4 Sonderfall: Bitflags

In vielen Fällen ist es so, dass ein Wert vom Datentyp WORD so interpretiert werden muss, dass jedes einzelne Bit eine eigene Bedeutung hat. Jedes Bit zeigt an, ob eine bestimmte Eigenschaft vorhanden ist oder nicht. Es ist wie eine Flagge (englisch: flag), die gesetzt sein kann (das Bit ist 1) oder nicht (das Bit ist Null). In dieser Situation sagt man, der Wert enthält **Bitflags**. Alternativ wird auch der Begriff **Flagbits** verwendet.

Ein Beispiel ist die globale Variable printFont.style. Die einzelnen Bits enthalten jeweils die Information ob der Text zum Beispiel **fett** (Bit TS_BOLD gesetzt), unterstrichen (TS_UNDERLINE gesetzt) oder *kursiv* (Bit TS_ITALIC gesetzt) ausgegeben werden soll. Sind alle drei Bits gesetzt, so wird der Text ***fett kursiv und unterstrichen*** ausgegeben.

Ein anderes Beispiel ist die Instancevariable csFeatures des ColorSelector-Objekts. Sie enthält für jedes UI-Element, dass der ColorSelector darstellen kann, ein Bit, das angibt, ob dieses UI-Element gezeigt oder verborgen werden soll. Gerade bei Instancevariablen und UI-Objekten kommen Bitflags relativ häufig vor.

Die Herausforderung bei Bitflags besteht darin, dass einzelne Bits gesetzt, zurückgesetzt oder angefragt werden müssen, ohne dass die anderen Bits beeinflusst werden. Hier helfen uns die logischen Operatoren (NOT, AND, OR, XOR) weiter.

Um ein Bit zu setzen verwenden wir die Operation **OR**. Das Ergebnis einer OR-Operation ist wahr, wenn mindestens einer der Operanden wahr ist.

```
neueBitFlags  = alteBitFlags OR bitsZuSetzen
```

Beispiele zum Setzen von Bits mit OR

alte BitFlags	Bits zu Setzen	neue Bitflags
0000	1000	1000
1000	1000	1000
1000	1001	1001
0011	1001	1011

Beispiel: Setzen des Bits TS_BOLD in der globalen Variablen printFont.style. Die Textausgabe erfolgt anschließend fett, egal ob sie vorher schon fett erfolgte oder nicht.

```
printFont.style  = printFont.style OR TS_BOLD
```

Es ist möglich, mehrere Bits gleichzeitig zu setzen. Das folgende Codefragment stellt sicher, dass sowohl das Bit TS_BOLD als auch das Bit TS_ITALIC gesetzt sind.

```
printFont.style  = printFont.style OR TS_BOLD OR TS_ITALIC
```

Um ein Bit zu zurückzusetzen (zu löschen) verwenden wir die Operatoren **AND** und **NOT**. Zuerst invertiert die NOT-Operation alle zu löschenden Bits, dann setzt die AND-Operation die zum Löschen vorgesehenen Bits auf Null.

```
neueBitFlags  = alteBitFlags AND ( NOT bitsZuLöchen )
```

Die Klammern sind eigentlich nicht nötig, da die NOT-Operation höher priorisiert ist, als die AND-Operation. Es wird aber dringend empfohlen bei Verwendung von mehreren logischen Operatoren innerhalb einer Anweisung Klammern zu setzen. Der Compiler vertut sich niemals mit der Hierarchie, der Programmierer schon.

Beispiele zum Löschen von Bits mit AND und NOT

BitFlags	Bits zu Löschen	negierte Bits (nach NOT)	Ergebnis
0000	0001	1110	0000
1111	0001	1110	1110
1101	1000	0111	0101
1100	1001	0110	0100
1011	1111	0000	0000

Beispiel: Löschen des Bits TS_BOLD in der globalen Variablen printFont.style. Die Textausgabe erfolgt anschließend nicht fett, egal ob sie vorher fett erfolgte oder nicht.

```
printFont.style  = printFont.style AND ( NOT TS_BOLD )
```

Es ist möglich, mehrere Bits gleichzeitig zu löschen. Dazu müssen die zu löschenden Bits zunächst OR-Verknüpft werden. In diesem Fall ist **notwendig** um die zu löschenden Bit eine **Klammer** zusetzen, weil OR niedriger priorisiert ist als NOT. Der Compiler gibt daher eine Warnung aus, wenn wir die Klammern vergessen.
Das folgende Codefragment stellt sicher, dass sowohl das Bit TS_BOLD als auch das Bit TS_ITALIC gelöscht sind.

```
printFont.style  = printFont.style AND \
                ( NOT ( TS_BOLD OR TS_ITALIC ) )
```

Um ein Bit zu invertieren verwenden wir die Operation **XOR**. Das Ergebnis einer XOR-Operation ist nur dann wahr, wenn genau einer der Operanden wahr ist. Sind beide wahr ist das Ergebnis falsch.

```
neueBitFlags  = alteBitFlags XOR bitsZuInvertieren
```

Beispiele zum Invertieren von Bits mit XOR

alte BitFlags	Bits zu Invertieren	neue Bitflags
0000	1000	1000
1000	1000	0000
1000	1001	0001
0011	1001	1010
1011	1111	0100

Beispiel: Invertieren des Bits TS_UNDERLINE in der globalen Variablen printFont.style. Die Textausgabe erfolgt anschließend unterstrichen, wenn sie vorher normal erfolgte. War sie vorher unterstrichen erfolgt sie jetzt normal.

```
printFont.style  = printFont.style XOR TS_UNDERLINE
```

Es ist möglich, mehrere Bits gleichzeitig zu invertieren. Dazu müssen die zu invertierenden Bits zunächst OR-Verknüpft werden. In diesem Fall ist **notwendig** um die zu löschenden Bit eine **Klammer** zusetzen, weil OR niedriger priorisiert ist als NOT. Der Compiler gibt daher eine Warnung aus, wenn wir die Klammern vergessen.
Das folgende Codefragment schaltet sowohl das Bit TS_BOLD als auch das Bit TS_ITALIC um.

```
printFont.style  = printFont.style XOR ( TS_BOLD OR TS_ITALIC )
```

Um ein Bit abzufragen verwenden wir die Operation **AND**. Das Ergebnis einer AND-Operation ist wahr, wenn beide Operanden wahr sind.

```
ergebnis  = bitFlags AND bitsZuTesten
```

Beispiele zum Abfragen von Bits mit AND

BitFlags	Bits zu Testen	Ergebnis
0000	1000	0000
1000	1000	1000
0111	1000	0000
0110	1001	0000
1111	1001	1001
1100	1001	1000

In der letzten Zeile der Tabelle ist zu sehen, dass das Ergebnis ungleich Null ist, obwohl nur eins der zu testenden Bits gesetzt ist. In einer IF-Anweisung wird aber jeder Wert ungleich Null als wahr angesehen. Deswegen müssen wir beim Testen von mehreren Bits unterscheiden, ob es uns reicht, dass eins der zu prüfenden Bits gesetzt ist oder ob alle Bits gesetzt sein müssen.

Beispiel: Testen, ob das Bit TS_BOLD in der globalen Variablen printFont.style gesetzt ist. Der THEN-Zweig wird ausgeführt, wenn das Bit gesetzt ist, egal wie die anderen Bits aussehen.

```
IF printFont.style AND TS_BOLD THEN
  ....
END IF
```

Abfrage von mehreren Bits

Wenn es ausreicht zu wissen, ob eins der zu prüfenden Bits gesetzt ist, reicht eine einfache Abfrage mit AND aus (siehe auch Tabelle vorn). In der Zahl 3 sind die Bits Null ($2^0 = 1$) und Eins ($2^1 = 2$) gesetzt. Das folgende Codefragment fragt ab, ob in der Variablen A das Bit Null **oder** das Bit Eins (**oder beide**) gesetzt sind.

```
IF A AND 3 THEN
  ....
END IF
```

Beispieltabelle für das Codefragment oben

Variable A	Bits zu Testen	Ergebnis	THEN Zweig ausführen?
0000 (= 0)	0011	0000	nein
0001 (= 1)	0011	0001	ja
0010 (= 2)	0011	0010	ja
0011 (= 3)	0011	0011	ja
0100 (= 4)	0011	0000	nein
0101 (= 5)	0011	0001	ja

Wenn wir sicherstellen wollen, dass alle zu prüfenden Bits gesetzt sind müssen wir das Ergebnis der AND-Verknüpfung mit den zu prüfenden Bits vergleichen. Das folgende Codefragment zeigt das wieder am Beispiel der Bits Null und Eins (Vergleichswert: $2^1 + 2^0 = 3$). Die **Klammern** sind **erforderlich**, weil Vergleiche höher priorisiert sind als die AND-Operation!

```
IF ( A AND 3 ) = 3 THEN
  ....
END IF
```

Beispieltabelle für das Codefragment oben. Beachten Sie die letzte Zeile!

Variable A	Bits zu Testen	A AND 3	THEN Zweig ausführen?
0000 (= 0)	0011	0000	nein
0001 (= 1)	0011	0001	nein
0010 (= 2)	0011	0010	nein
0011 (= 3)	0011	0011	ja
0100 (= 4)	0011	0000	nein
0111 (= 7)	0011	0011	ja

2.3.6 Schnelle Mathematik mit WWFixed

Der hohen Genauigkeit und dem großen Wertebereich des Zahlenbereichs REAL, mit der R-BASIC im Normalfall rechnet, steht als Nachteil eine höhere Rechenzeit gegenüber. R-BASIC bietet deswegen mit den Datentyp **WWFixed** (Word-Word-Fixed) die Möglichkeit, von der schnelleren Ganzzahlarithmetik zu profitieren. Da der Zeitbedarf zum Lesen und Schreiben von Variablen sowie der Analyse des mathematischen Terms gleich bleibt hängt der erreichbare Geschwindigkeitsvorteil etwas von der Situation ab. Reine WWFixed-Berechnungen laufen ca. 25% bis 30% schneller. Wenn Sie innerhalb eines WWFixed-Ausdrucks auf Real-Variablen zugreifen verringert sich der Laufzeitvorteil wegen der notwendigen Konvertierungen geringfügig.

Im Folgenden werden "normale" numerische Ausdrücke mit den Datentypen REAL, Byte, Word, DWord, Integer und LongInt als "Real"-Ausdrücke bezeichnet. Hier rechnet R-BASIC immer mit 10-Byte Real-Zahlen. Im Gegensatz dazu stehen die "WWFixed"-Ausdrücke, in denen R-BASIC die schnellere Ganzzahlarithmetik verwendet.

Konvertierungsfunktionen

WWFixed-Werte sind nur begrenzt zuweisungskompatibel mit den anderen numerischen Datentypen. Das ist Absicht, damit nicht durch die gemischte Verwendung von Real- und WWFixed-Werten der Performancegewinn unabsichtlich aufgebraucht wird.
Wenn Sie das Ergebnis einer WWFixed-Rechnung in einem Real-Ausdruck verwenden wollen oder einen Real-Wert in eine WWFixed-Rechnung einbinden möchten müssen Sie in meistes die folgenden schnellen Konvertierungsroutinen benutzen. Es gibt jedoch auch Ausnahmen, die weiter unten beschrieben sind.

Funktion	Bedeutung
MakeFixed(x)	Rechnet einen REAL-Wert in einen WWFixed-Wert um.
FixToReal(x)	Rechnet einen WWFixed-Wert in einen REAL-Wert um.
FixToWord(x)	Rechnet einen WWFixed-Wert in einen REAL-Wert um. Die Zahl wird zuerst gerundet (x=ROUND(x)), das Ergebnis wird als Word-Wert interpretiert. Negative Zahlen werden so zu Werten größer als 32767.

Beispiele

```
DIM f1, f2 AS WWFixed
DIM a as REAL
  f1 = MakeFixed( MyNumberObj.value )
  f2 = MakeFixed( ASC( "A" ) )
  LINE 0, 0, FixToWord( f1 ), FixToWord ( f1 + f2 )
  MyNumberObj.value  = FixToReal ( f1 * f2 )
  a = FixToReal (f1/f2)
```

Der Datentyp WWFixed besteht intern aus einem DWord. Das höherwertige Word wird als vorzeichenbehafteter ganzzahliger Anteil interpretiert (Wertebereich: Integer), das niederwertige Word ist der gebrochene Anteil. Ein Bit entspricht damit einem Wert von $1{,}526 \cdot 10^{-5}$. Daraus ergibt sich ein Wertebereich von

```
– 32768.0000 <= x <= 32767.99999
```

mit einer Genauigkeit von 4 Stellen nach dem Komma. Diese Genauigkeit ist für viele Anwendungen völlig ausreichend.

Sie können Berechnungen mit WWFixed-Werten genau so programmieren, wie mit jedem anderen numerischen Datentyp. Es gibt jedoch drei **Ausnahmen**:

- Die Verwendung von einigen mathematischen Funktionen ist nicht zulässig. In den Tabellen unten finden Sie die zugelassenen Funktionen.
- Das Lesen von numerischen Instancevariablen ist nicht zulässig.
- Die Verwendung der Operationen MOD und ^ (x-hoch-y) ist nicht möglich.

Sie müssen in diesen Fällen die Konvertierungsfunktion MakeFixed( ) benutzen.

Der Compiler erwartet einen WWFixed-Ausdruck in zwei Fällen

1. Wir haben eine Zuweisung zu einer WWFixed Variablen.
2. Wir haben eine Sub oder Function, die einen WWFixed Parameter erwartet.

Zulässig für WWFixed-Ausdrücke sind:

- Zahlen (in beliebiger Schreibweise, auch mit binär, hexadezimal und mit Exponent)
- Numerische BASIC-Konstanten (wie z.B. PI oder GREEN) und selbst definierte Konstanten (Anweisung: CONST). Der R-BASIC Compiler rechnet den Wert automatisch in eine WWFixed-Zahl um.
- Grundrechenarten und Klammern
- Variablen vom Typ WWFixed
- Funktionen (auch selbst definiert) mit dem Rückgabetyp WWFixed
- Logische Operatoren
- Vergleichsoperatoren

Um den extensiven Gebrauch der Konvertierungsfunktionen zu vermeiden sind **zusätzlich** folgende Elemente zulässig:

- Variablen der anderen numerischen Datentypen. Die Werte werden von R-BASIC automatisch in den Datentyp WWFixed konvertiert.
- Die in den Tabellen unten ausgewiesenen numerischen Funktionen. R-BASIC verwendet in diesem Fall nicht den gleichen Code wie für Real-Funktionen, sondern spezielle, für die Verwendung von WWFixed-Werten angepasste Funktionen.

Berechnungen mit WWFixed Werten führt R-BASIC mit der schnellen 32 Bit Ganzzahlarithmetik durch. Es erfolgt **keine Fehlerprüfung**. Das bedeutet:

- Überträge werden nicht erkannt. Multipliziert oder addiert man zum Beispiel zwei WWFixed-Zahlen und das Ergebnis ist größer als 32768, so werden die höherwertigsten Bits abgeschnitten. Das Ergebnis ist selbst für erfahrene Programmierer schwer vorherzusehen. Häufig entstehen sogar negative Zahlen.

- Division durch Null oder die Wurzel aus einer negativen Zahl führt nicht zu einem Fehler, sondern zu einem zufälligen Wert.
- Zuweisung von Werten außerhalb des gültigen Wertebereichs führt häufig zu "unerwarteten" Ergebnissen.

Prüfen Sie vor der Verwendung von WWFixed-Berechnungen unbedingt, ob die Berechnungen den zulässigen Wertebereich nicht überschreiten können. Die Verletzung dieser Regel kann zu schwer auffindbaren Fehlern führten!

Rechenoperationen

Innerhalb von WWFixed-Ausdrücken sind die folgenden Rechenoperationen erlaubt:

Operationen	Bedeutung
+, –, *, /, (,)	Grundrechenarten, Klammern
AND, OR, XOR, NOT	bitweise logische Operationen
<, <=, =, >, >=, <>	Vergleiche

Die Operationen MOD (Division mit Rest) und ^ (x-hoch-y) sind innerhalb von WWFixed-Ausdrücken nicht zulässig.

Beispiele:

```
DIM a, b, c   AS WWFixed
CONST f = PI/2
  a = 12.5
  b = 180 * f * a
  c = 12 *( a + b) / PI
  PRINT "Ergebnis = "; FixToReal(c)
```

```
DIM a, b, c   AS WWFixed
  b = a OR 7
  c = a AND 2
```

Die logischen Operationen liefern WWFixed-Werte, wenn Sie in WWFixed-Ausdrücken verwendet werden. Auch hier müssen Sie die Konvertierungsfunktion FixToReal benutzen, wenn Sie logische Operationen mit WWFixed-Werten innerhalb einer Entscheidungsanweisung verwenden.

Beispiele:

```
DIM fa, fb AS WWFixed
  fa = fb OR 4                          ' einfache Zuweisung
  IF FixToReal( fa AND 2 ) THEN ...
  WHILE FixToReal( fa AND 1 )
    ....
    WEND
```

Die Vergleichsoperationen liefern TRUE (numerischer Wert: –1) oder FALSE (numerischer Wert: 0). Das Ergebnis ist ein WWFixed-Wert. Da IF-Anweisungen (und andere Entscheidungsoperationen wie WHILE) einen Real-Ausdruck erwarten müssen Sie hier die Konvertierungsfunktion FixToReal benutzen.

Beispiele:

```
DIM fa, fb AS WWFixed
  IF FixToReal( fa > fb ) THEN ...
  WHILE FixToReal( fa < 0 )
    ....
    WEND
```

Einfache mathematische Funktionen:

Um den gehäuften Aufruf der Konvertierungsfunktionen zu vermeiden, können innerhalb von WWFixed-Ausdrücken die folgenden einfachen numerischen Funktionen verwendet werden. Der Compiler erkennt dabei, dass es sich um einen WWFixed-Ausdruck handelt und compiliert den Aufruf einer für WWFixed optimierten Routine. Dadurch profitieren diese Funktionen ebenfalls vom Geschwindigkeitsvorteil der WWFixed-Mathematik. Syntaktisch unterscheidet sich die Verwendung der Funktionen nicht von ihren Real-Versionen. Ausnahme ist die Round-Funktion. Hier ist kein zweiter Parameter zulässig und bei x.5 wird immer nach unten gerundet.

Funktion	Bedeutung
Abs(x)	Absoluter Betrag von x: |x|
Sgn(x)	Signum-Funktion (Vorzeichen-Funktion) Liefert –1 (negativ), 0 oder +1 (positiv)
Int(x)	Liefert die nächst kleinere ganze Zahl, d.h. es wird nach unten gerundet: Int(x) ≤ x
Trunc(x)	Kürzt x auf seinen ganzahligen Anteil, d.h. es wird Richtung Null gerundet. Trunc(x) und Int(x) unterschieden sich bei negativen x.
Frac(x)	Liefert den gebrochenen Anteil von x, d.h. die Nachkommastellen. Das Ergebnis ist immer positiv.
Round(x)	Rundet x auf die nächste ganze Zahl
SizeOf(x)	Berechnet den Speicherbedarf einer Variablen oder eines Datentyps. Der Wert wird vom Compiler ermittelt und als Zahl gespeichert.

Beispiele:

```
DIM a, b, c, d as WWFixed
  a = 12.3
  b = INT(a)
  c = FRAC(a)
  d = 4 * ROUND(a) - b * SizeOf(WWFixed)
```

Höhere mathematische Funktionen:

Für WWFixed-Ausdrücke sind die folgenden höheren Funktionen definiert. R-BASIC verwendet zur Berechnung dieser Funktionen die schnelle Ganzzahlarithmetik.

Funktion	Bedeutung
FixSqr(x)	Quadratwurzel von x
FixSin(x)	Sinus von x, x im Gradmaß
FixCos(x)	Cosinus von x, x im Gradmaß
FixTan(x)	Tangens von x, x im Gradmaß
FixAsn(x)	ArcusSinus von x - Umkehroperation zu Sinus

Für das Argument der Winkelfunktionen sind auch Werte außerhalb des Bereichs 0 bis 360° zulässig. Das gilt auch für negative Werte.
FixAsn und FixSqr arbeiten nur für positive Argumente korrekt. Negative Argumente führen zu fehlerhaften Ergebnissen, aber nicht zu einer Fehlermeldung.

Alle in der Tabelle oben angegeben Funktionen sind auch innerhalb von Real-Ausdrücken zulässig. Der R-BASIC Compiler erkennt den Aufruf dieser Fixed-Funktionen und compiliert automatisch den Aufruf der Konvertierungsfunktion FixToReal( ).

Die Winkelfunktionen für WWFixed-Ausdrücke unterscheiden sich von den Winkelfunktionen der Real-Ausdrücke dadurch, dass sie das Argument im Gradmaß (der Vollkreis hat 360°) erwarten. Die Real-Winkelfunktionen erwarten das Argument dagegen im Bogenmaß (ein Vollkreis entspricht 2π). Wenn Sie das Argument im Bogenmaß haben brauchen Sie es nicht selbst ins Gradmaß umzurechnen. Stattdessen können Sie innerhalb von WWFixed-Ausdrücken die in der folgenden Tabelle angegebenen Real-Winkelfunktionen direkt verwenden. R-BASIC erkennt, dass es sich um einen WWFixed-Ausdruck handelt, rechnet das Argument automatisch ins Bogenmaß um und ruft dann die entsprechende Fixed-Funktion auf. Da ist sogar noch etwas schneller als die manuelle Umrechnung.

Funktion	Bedeutung
SIN(x)	Sinus von x, x im Bogenmaß
COS(x)	Cosinus von x, x im Bogenmaß
TAN(x)	Tangens von x, x im Bogenmaß
ASN(x)	ArcusSinus von x - Umkehroperation zu Sinus

Beispiele:

```
DIM x, y AS WWFixed
  y = 50 * FixSin(x) + 150          ' x im Gradmaß
  y = 1/TAN(x)                      ' Cotangens, x im Bogenmaß
```

2.3.7 Exkurs: Vergleiche innerhalb numerischer Ausdrücke

Achtung! Die folgenden Ausführungen richten sich an fortgeschrittene Programmierer. Sie sind daher möglicherweise etwas abstrakt. Ihr Verständnis kann hilfreich sein, ist aber zur Anwendung innerhalb von Entscheidungsausdrücken nicht unbedingt erforderlich.

Intern behandelt R-BASIC Vergleichsausdrücke und logische Ausdrücke als Zahlen. Das hat Konsequenzen: Jeder Vergleich liefert entweder TRUE (wahr, –1) oder FALSE (falsch, Null). Es ist deshalb möglich, das Ergebnis eines Vergleichs in einer numerischen Variablen abzuspeichern.

Beispiele (DIM A, B, Y vorausgesetzt):

```
Y = A > 0
```

Der Compiler erkennt die Zuweisung "Y =" und berechnet den numerischen Ausdruck auf der rechten Seite. Da "A > 0" ein gültiger numerischer Ausdruck ist, der TRUE (–1) oder FALSE (0) liefern kann, wird Y der Wert 0 oder –1 zugewiesen, je nachdem ob A größer Null ist, oder nicht.

```
Y = Name$ = "Paul"
```

Der Compiler erkennt wieder die Zuweisung "Y =" und berechnet den rechten Ausdruck (den Vergleich Name$ = "Paul"), der wiederum TRUE (–1) oder FALSE (0) ergeben kann.

In diesem Zusammenhang sind Vergleichsausdrücke nur spezielle numerische Ausdrücke:

Beispiel 1:

```
Y = (B < 7)  AND  (Name$ = "Paul")
```

Der Compiler berechnet die Ausdrücke (B < 7) und (Name$ = "Paul"), die jeweils TRUE (–1) oder FALSE (0) ergeben. Anschließend wird die logische Verknüpfung AND ausgeführt. AND ist zwar eine bitweise Funktion, aber das Zusammenspiel funktioniert, da in TRUE alle 16 Bit gesetzt sind, während in FALSE alle 16 Bit Null sind. Y wird TRUE, wenn beide Bedingungen erfüllt sind und FALSE, wenn mindestens eine Bedingung nicht erfüllt ist.

Beispiel 2:

```
Y = ( A > 5 )  AND  7       ' Liefert 0 oder 7, je nach A
```

2.4 Arbeit mit Strings

2.4.1 Bearbeiten von Strings

Zeichenketten wie z.B. "Hallo Welt" werden in BASIC als Strings bezeichnet. Die Verarbeitung von Strings ist eine der grundlegenden Fähigkeiten von R-BASIC. Die Grundlagen zur Verwendung von Strings und von Stringvariablen finden Sie im Kapitel 2.2.3 (Stringtypen und Stringausdrücke).

Left$

Die Funktion Left$(A$, N) (Left - links) liefert die N ersten (linken) Zeichen des Strings A$.

Syntax:	**<stringVar>** = **Left$**(**A$**, **N**)	
Parameter:	A$:	ein Stringausdruck
	N:	numerischer Ausdruck, Anzahl der Zeichen

Beispiel:

```
L$ = Left$("Paulchen N." , 4)    ' Entspricht L$ = "Paul"
L$ = Left$(L$ , 1)               ' Macht aus "Paul" ein "P"
```

Right$

Die Funktion Right$(A$, N) (Right - rechts) liefert die N letzten (rechten) Zeichen des Strings A$.

Syntax:	**<stringVar>** = **Right$**(**A$**, **N**)	
Parameter:	A$:	ein Stringausdruck
	N:	numerischer Ausdruck, Anzahl der Zeichen

Beispiel:

```
R$ = Right$("Paulchen N." , 4)       ' Entspricht R$ = "n N."
R$ = Right$ (Left$("ABCDEF" , 4) , 2 ) ' liefert "CD"
```

Mid$

Die Funktion Mid$(A$, P, N) (Middle - Mitte) liefert N Zeichen ab der Position P in einem String.

Formate:	**<stringVar>** = **Mid$**(**A$**, **P**, **N**)	
	<stringVar>= **Mid$**(**A$**, **P**)	
Parameter:	P:	Erste Zeichenposition, die kopiert werden soll
	N:	Anzahl der Zeichen die kopiert werden sollen
		ohne N: Alle restlichen Zeichen werden kopiert

Beispiele:

```
X$ = Mid$("OTTOKAR", 3, 4 )      ' liefert "TOKA"
X$ = Mid$("OTTOKAR", 5)          ' liefert "KAR"
```

Anmerkung: Ist der String A$ zu kurz, werden entsprechen weniger Zeichen geliefert. Ist der String A$ kürzer als der Parameter P erfordert, liefert Mid$ einen leeren String.

Trim$

Die Funktion Trim$(A$ [,mode]) entfernt Leerzeichen und Tabs am Anfang und / oder am Ende der Zeichenkette.

Syntax: **<stringVar>** = **Trim$(A$** [,**mode**])
Parameter: mode: numerischer Ausdruck, bestimmt ob Leerzeichen und Tabulatoren am Anfang (mode = 1), Am Ende (mode =2) oder beiden (mode = 3, Defaultwert) entfernt werden sollen.
A$: Stringausdruck, der bearbeitet werden soll.

Beispiel:

```
X$ = Trim$("  Paul  ")       ' liefert "Paul"
X$ = Trim$("  Paul  ", 1)    ' liefert "Paul  "
X$ = Trim$("  Paul  ", 2)    ' liefert "  Paul"
```

ReplaceStr$

Die Funktion ReplaceStr$(s$, a$, b$) ersetzt jedes Auftreten des Strings a$ in s$ durch b$.

Syntax: <stringVar> = ReplaceStr$(s$, a$, b$)
<stringVar>: Stringvariable
s$: String, der durchsucht werden soll
a$: String, der ersetzt werden soll
b$: String, der a$ ersetzen soll
b$ darf genauso lang, länger oder kürzer als a$ sein.
Ist b$ ein Leerstring wird jedes Auftreten von a$ gelöscht

Beispiele:

Anweisung	Ergebnis
ReplaceStr$ ("Hallo", "a", "e")	"Hello"
ReplaceStr$ ("Hallo", "al", "(xyz)")	"H(xyz)lo"
ReplaceStr$ ("Hallo", "ll", "")	"Hao"
ReplaceStr$ ("Hallo Welt!", "l", "-X-")	"Ha-X--X-o We-X-t!"
ReplaceStr$ ("12.75 Euro", ".", ",")	"12,75 Euro"

String$

Die Funktion String$(N, A$) vervielfacht Zeichenkettenausdrücke.

Syntax: **<stringVar>** = **String$**(**N**, **A$**)
Parameter: N: numerischer Ausdruck, Anzahl der Vervielfachungen
A$: Stringausdruck, der vervielfacht werden soll

Beispiel:

```
  X$ = String$(4, "X")                  ' liefert "XXXX"
  X$ = String$(2, Left$("KOMA", 2) )    ' liefert "KOKO"
```

InStr

Die Funktion InStr(A$, B$) (d.h. In-String) ermittelt die Position, ab welcher A$ in B$ enthalten ist.

Syntax: **<numVar>** = **InStr**(**A$**, **B$**)
Parameter: A$: String-Ausdruck: der zu findende String
B$: String-Ausdruck: String, der A$ enthalten soll

Beispiel:

```
DIM Anz
  Anz = InStr("ul", "Paula")           ' liefert 3
  Anz = InStr("lala", "Paula")         ' liefert Null
```

Hinweise:
- Groß- und Kleinbuchstaben werden unterschieden
- Ist der String nicht oder nicht vollständig enthalten, liefert InStr den Wert Null.
- Ist einer der beiden Strings ein Leerstring (""), liefert InStr den Wert Null.

LEN

Die Funktion LEN (Length - Länge) liefert die Länge des Strings, d.h. die Anzahl der enthaltenen Zeichen.

Syntax: **<numVar>** = **LEN**(**A$**)
Parameter: A$ String-Ausdruck

Beispiel:

```
DIM A
  A = LEN("Paula")        ' liefert 5
  A = LEN("")         ' liefert Null
```

CountStr

Die Funktion CountStr(A$, B$) zählt, wie oft A$ in B$ enthalten ist.

Syntax: **<numVar>** = **CountStr**(A$, B$)
Parameter: A$: String-Ausdruck: der zu findende String
B$: String-Ausdruck: String, der A$ enthalten soll

Beispiel:

```
DIM N
  N = CountStr("ul", "Paula")      ' liefert 1
  N = CountStr("a", "Paula")       ' liefert 2
  N = CountStr("aha", "Hahaha")    ' liefert 1
```

Hinweise:
- Groß- und Kleinbuchstaben werden unterschieden
- Ist der String nicht oder nicht vollständig enthalten, liefert CountStr den Wert Null.
- Ist einer der beiden Strings ein Leerstring (""), liefert CountStr den Wert Null.
- Beachten Sie Beispiel 3! Buchstaben, die bereits gefunden wurden, werden nicht noch einmal berücksichtigt.

Stringoperation +

Die Operation + verbindet zwei Strings.

Syntax: **<stringVar>** = **A$** + **B$**
Parameter: A$, B$: Beliebige Stringausdrücke

Beispiel:

```
  A$ = "Paul" + " " + "Müller"         ' liefert "Paul Müller"
  A$ = "->" + Left$("Paul", 2) + "<-"  ' liefert "->Pa<-"
```

Tipp:
- Klammern, z. B. ("Paul" + " ") + "Müller", sind möglich, aber nicht erforderlich.

GetTextWitdh, GetTextHeight

GetTextWidth berechnet die Breite (in Pixeln), die ein String bei Ausgabe auf den Bildschirm benötigt. GetTextHeight berechnet die entsprechende Höhe (in Pixeln). Die beiden Routinen sind in allen Fontmodi (Fixed-, Block- und GEOS-Font-Modus, siehe Handbuch Spezielle Themen, Kapitel 2), anwendbar. Sie eignen sich zum Beispiel um einen Text zentriert an eine bestimmte Position zu drucken.

Syntax: **<numVar>** = **GetTextWidth** (**"Text"**)
<numVar> = **GetTextHeight** (**"Text"**)

Beachten Sie, dass im GEOS-Font-Modus (Routine FontSetGeos, siehe Handbuch Spezielle Themen, Kapitel 2.4) die Breite und Höhe wirklich nur den von den Buchstaben überdeckten Bereich umfassen. Der als Texthintergrund eingefärbte Bereich ist im Allgemeinen merklich größer. Das kann insbesondere dann verwirrend sein, wenn Sie einen Rahmen um einen Text zeichnen wollen.

Die folgende Routine zeichnet einen Rahmen um einen Text. Im GEOS-Font-Modus machen wir den Rahmen etwas breiter (4 Pixel) und höher (8 Pixel) und berücksichtigen, dass der eigentliche Buchstabe immer etwas rechts unterhalb der in Print atXY angegebenen Position gezeichnet wird. Die notwendigen Werte für die Verschiebung und die Vergrößerung des Rahmens hängen etwas vom eingestellten Font und der Schriftgröße ab.

```
SUB DrawTextFramed(t$ as String, x, y AS REAL)
DIM w, h

  Print atXY x, y; t$

  w = GetTextWidth(t$)
  h = GetTextHeight(t$)
  IF printfont.type = FT_GEOS THEN
    x = x - 1
    y = y + 2
    w = w + 4
    h = h + 8
  End IF

  Rectangle x-1, y-1, x+w, y+h, LIGHT_CYAN
END SUB
```

2.4.2 Vergleichen von Strings

Vergleichsoperatoren

Die Standard-Vergleichsoperatoren (<, <=, >, >=, =, <>) stehen auch für Zeichenketten (Strings) zur Verfügung. Die Strings werden dabei Zeichen für Zeichen verglichen, wobei ausschließlich die ASCII-Codes der einzelnen Zeichen berücksichtigt werden. Das heißt z.B., dass Umlaute entsprechend ihrer Position in der ASCII-Code-Tabelle (also noch hinter z) gewertet werden. Für lexikalisch korrekte Vergleiche verwenden Sie bitte die unten beschriebene Funktion CompStr.

Syntax:	**A$ < B$** **A$ <= B$** usw.
Parameter:	A$, B$ String-Ausdrücke, die verglichen werden sollen
Ergebnis:	Wahr (TRUE, numerischer Wert: –1) oder Falsch (FALSE, numerischer Wert: 0)

Beispiel:

```
IF A$ >= "Paul" THEN ...
IF A$ <> Left$("Paul", 2)  THEN ...
```

Hinweis: Statt A$ = B$ kann man für Vergleich auch ein doppeltes Gleichheitszeichen schreiben (A$ == B$). Das verbessert gelegentlich die Lesbarkeit.

Tipp:
String-Vergleiche arbeiten auch mit den logischen Operatoren zusammen. Da Vergleiche höher priorisiert sind, benötigt man hier meist keine Klammern.
Beispiele:

```
   IF  NAME$ = "Müller"  OR  NAME$ = "Meier"  THEN ...
   IF  A$ < B$  OR  A$ < C$  THEN ...
```

Tipp für Fortgeschrittene:
String-Vergleiche liefern als Ergebnis eine Zahl (Null oder –1), sie dürfen deshalb innerhalb von numerischen Ausdrücken vorkommen. Beispiel (Die Klammern sind nicht erforderlich, sie verbessern die Lesbarkeit.):

```
DIM A As Real
DIM Name$ as String
  A = (Name$ = "Müller")
        '  A wird –1 (wahr), wenn Name$ = "Müller" ist.
        ' andernfalls wir A Null
```

CompStr

Die Funktion CompStr (Compare Strings, d.h. vergleiche Zeichenketten) vergleicht zwei Strings entsprechend ihrer Anordnung im Wörterbuch. Dabei gelten folgende Regeln:

- Umlaute reihen sich ein
- Großbuchstaben stehen bei ansonsten gleichen Strings vor den Kleinbuchstaben. Beispiel: "Schwimmen" steht vor "schwimmen"
- Kürzere Strings stehen bei Gleichheit mit längeren Strings vorn. Beispiel: "Paul" steht vor "Paula".

Syntax:	**<numVar>** = **CompStr**(**A$**, **B$**)	
Parameter:	A$, B$	String-Ausdrücke, die verglichen werden sollen
Ergebnis:	–1	wenn A$ < B$, d.h. A$ steht vor B$ im Wörterbuch
	0	wenn A$ = B$, d.h. A$ und B$ stehen an gleicher Stelle im Wörterbuch. Dann sind beide Strings identisch.
	+1	wenn A$ > B$, d.h. A$ steht nach B$ nach Wörterbuch

Beispiel:

```
Dim V as Real
  IF CompStr(A$, B$) > 0   THEN ..
  V = CompStr(A$, B$)
```

Achtung! Die Funktion verwendet länderspezifische Regeln. Es ist möglich, dass auf anderen, insbesondere auf nicht deutschen PC/GEOS-Systemen, andere Regeln bezüglich der Anordnung im Wörterbuch gelten. CompStr verwendet die auf dem jeweiligen PC/GEOS-System geltenden Regeln.

2.4.3 Konvertierungsfunktionen

Konvertierung Zahl zu String

Bin$

Die Funktion Bin$ (Binär Konvertierung) wandelt eine Ganzzahl in ihre binäre Darstellung (zur Basis 2) um. Nichtganzzahlige Argumente x werden gerundet. Negative Zahlen werden ebenfalls korrekt behandelt.

Syntax: **<stringVar>** = **Bin$**(**x** [,**min** [,**max**]])
Parameter: x: numerischer Ausdruck, Zahlenbereich DWord (32 Bit)
min: optional: Mindestanzahl auszugebender Binärziffern. Es werden bei Bedarf führen Nullen hinzugefügt.
max: optional: Maximalzahl auszugebender Binärziffern. Führende Stellen werden gerundet.
min und max dürfen im Bereich von 1 bis 32 liegen

Beispiel:

```
A$ = Bin$(5)              ' A$ = "101"
A$ = Bin$(18)             ' A$ = "10010"
A$ = Bin$(5, 4)           ' A$ = "0101"
A$ = Bin$(18, 4, 4)       ' A$ = "0010"
                          ' das fünfte Bit wird ignoriert
```

Tipp: Setzen Sie min = max für eine feste Stellenzahl (siehe letztes Beispiel)

Chr$

Die Funktion Chr$(x) (Character - Zeichen) liefert das Text-Zeichen, das zum ASCII-Code x gehört. Chr$(0) liefert einen leeren String.

Syntax: **<stringVar>** = **Chr$**(**x**)
Parameter: x: numerischer Ausdruck

Beispiel:

```
A$ = Chr$(65 + 1)           ' Entspricht A$ = "B"
```

Innerhalb von Strings kann man statt der Funktion Chr$(x) auch einen Backslash, gefolgt von bis zu drei Ziffern, verwenden. Folgende Zeilen sind daher gleichwertig:

```
A$ = "a"  + Chr$(180) + "b"
A$ = "a\180b"
```

Hex$

Die Funktion Hex$ (Hexadezimal Konvertierung) wandelt eine Ganzzahl in ihre hexadezimale Darstellung (zur Basis 16) um. Nichtganzzahlige Argumente x werden gerundet. Negative Zahlen werden ebenfalls korrekt behandelt.

Syntax: **<stringVar>** = **Hex$**(**x** [,**min** [,**max**]])
Parameter: x: numerischer Ausdruck, Zahlenbereich DWord (32 Bit d.h. 8 Hex-Stellen)
min: optional: Mindestanzahl auszugebender Hex-Ziffern. Es werden bei Bedarf führen Nullen hinzugefügt.
max: optional: Maximalzahl auszugebender Hex-Ziffern. Führende Stellen werden gerundet.
min und max dürfen im Bereich von 1 bis 8 liegen

Beispiele:

```
A$ = Hex$(11)           ' A$ = "B"
A$ = Hex$(11, 2)           ' A$ = "0B"
A$ = Hex$(764, 2)          ' A$ = "2FC"
A$ = Hex$(764, 2, 2)       ' A$ = "FC"
A$ = Hex$(764, 4, 4)       ' A$ = "02FC"
```

Tipp: Setzen Sie min = max für eine feste Stellenzahl (siehe letztes Beispiel)

Str$

Die Funktion Str$(x) (String - Zeichenfolge) konvertiert eine Zahl in eine Zeichenkette, genau so, als ob Sie die Zahl mit PRINT auf den Bildschirm ausgeben. Als Dezimaltrennzeichen wird immer der Punkt '.' verwendet.

Syntax: **<stringVar>** = Str$(**x**)
Parameter: x: numerischer Ausdruck

Beispiel:

```
A$ = Str$( 12+3 )       ' Entspricht A$ = " 15 "
```

Hinweis: Das Zahlenformt der konvertierten Zahl kann mit der System-Variablen numberFormat beeinflusst werden.

StrLocal$

Die Funktion StrLocal$(x) (String Local) konvertiert eine Zahl in eine Zeichenkette unter Verwendung der lokal (auf dem aktuellen Rechner) gültigen Einstellungen für Dezimal- und Tausender-Trennzeichen. Verwenden Sie diese Konvertierungs-Funktion, wenn Sie dem Nutzer seine "gewohnte" Zahlendarstellung präsentieren wollen.

Syntax: **<stringVar>** = StrLocal$(**x**)
Parameter: x: numerischer Ausdruck

Beispiel:

```
A$ = StrLocal$(9.82347E5 )  ' Liefert auf den meisten
                            ' deutschen Computern " 98.234,7"
```

Achtung:
Das Ergebnis dieser Konvertierungsfunktion hängt von den Einstellungen des aktuellen Computers ab. Sie sollten keine Annahmen über das Format der konvertierten Zahl machen.

Hinweis:
Das Zahlenformt der konvertierten Zahl kann mit der System-Variablen numberFormat beeinflusst werden.

Konvertierung String zu Zahl

ASC

Die Funktion ASC (von ASCII) liefert den ASCII-Code des ersten Zeichens des Strings. ASC("") (Leerstring) liefert Null.

Syntax: **<numVar>** = ASC(**A$**)
Parameter: A$: String-Ausdruck

Beispiel:

```
X = ASC("Auto")         ' liefert 65, den ASCII-Code von A
```

VAL

Die Funktion VAL (Value - Wert) wandelt eine Zeichenkette in die entsprechende Zahl um. Entspricht die Zeichenkette keiner Zahl, versucht VAL seine Aufgabe "so gut wie möglich" zu erfüllen und konvertiert so viele Zeichen wie es finden kann - findet es gar keine gültigen Zeichen, liefert es Null.

Syntax: **<numVar>** = VAL(**A$**)
Parameter: A$: String-Ausdruck

Beispiel:

```
X = VAL("-12.8")            ' Entspricht X = -12.8
X = VAL("Paul")             ' liefert Null.
```

ValLocal

Die Funktion ValLocal wandelt eine Zeichenkette in die entsprechende Zahl um. Dabei werden die lokal (auf dem aktuellen Computer) eingestellten Dezimal- und Tausendertrennzeichen erwartet. Verwenden Sie diese Konvertierungs-Funktion, wenn der Nutzer eine Zahl in der für ihn vertrauten Weise eingeben soll.

Syntax: **<numVar>** = ValLocal(**A$**)
Parameter: A$: String-Ausdruck

Beispiel:

```
X = ValLocal("12.824,2")    ' funktioniert für die meisten
                            ' deutschen Computer und liefert
                            ' Zwölftausendachthundertvierundzwanzig Komma Sieben
```

Achtung:
Ob die Zahl korrekt konvertiert wird, hängt von den Einstellungen des aktuellen Computers ab.

Zeichensätze konvertieren

Convert$

GEOS verwendet einen anderen Zeichensatz als DOS oder Windows, d.h. die ASCII-Codes für bestimmte Zeichen (ASCII-Code >= 128) unterscheiden sich und es gibt Zeichen in jedem Zeichensatz, die in den anderen Zeichensätzen nicht darstellbar sind. Die Funktion Convert$() übernimmt die Konvertierung zwischen den verschiedenen Zeichensätzen und ersetzt Codes, die im neuen Zeichensatz nicht darstellbar sind.

Syntax: **<stringVar>** = **Convert$**(**A$**, **mode** [, **replace**])
Parameter: A$: Stringausdruck, der konvertiert werden soll
mode: Bestimmt, zwischen welchen Zeichensätzen konvertiert werden soll. Siehe Tabelle unten.
replace: ASCII-Code des Ersatz-Zeichens, falls ein Zeichen nicht konvertierbar ist. 'replace' ist optional.
- Wird 'replace' nicht angegeben, wird das Standardzeichen '_' verwendet.
- Wird für 'replace' Null angegeben, werden nicht konvertierbare Zeichen gelöscht.

Für ’mode’ stehen folgende Werte zur Verfügung:

Wert	Bezeichnung	Konvertierung zwischen ...
0	(keine)	Zeichensatz nicht ändern
1	DOS_TO_GEOS	DOS nach GEOS
2	GEOS_TO_WIN	GEOS nach Windows (Codepage 1252, ANSI)
3	WIN_TO_GEOS	Windows (Codepage 1252, ANSI) nach GEOS
4	GEOS_TO_DOS	GEOS nach DOS (Codepage 437)
5	HTML_TO_GEOS	HTML-Codes für Umlaute und Sonderzeichen werden durch die entsprechenden GEOS Zeichen ersetzt. Tags wie werden nicht geändert, ’replace’ wird nicht verwendet. Zeichen mit einem Code über 127 werden durchgereicht (nicht geändert).
6	GEOS_TO_HTML	Jedes GEOS-Zeichen mit einem Code über 127, z.B. Umlaute, werden durch den entsprechenden HTML-Code ersetzt. ’Replace’ wird nicht verwendet.
7	HTML_TO_GEOS_BR	Wie HTML_TO_GEOS, allerdings wird jedes der Tags , , und </br> durch ein "CarriageReturn" (CR, Code 13 bzw "\r") ersetzt.
8	GEOS_TO_HTML_BR	Wie GEOS_TO_HTML. Zusätzlich wird vor jedem "CarriageReturn" (CR, Code 13 bzw "\r") ein HTML-Zeilenumbruch " " eingefügt.
9	UTF8_TO_GEOS	UTF-8 nach GEOS
10	GEOS_TO_UTF8	GEOS nach UTF-8

Jeder der Werte aus der Tabelle oben kann mit den folgenden Flags kombiniert werden, indem die Werte addiert oder logisch OR kombiniert werden. Außerdem können die Flags auch allein verwendet werden.

Wert	Bezeichnung	Wirkung
16	CRLF_TO_CR	Jedes Auftreten der Codefolge CR+LF (Codes 13 und 10) oder von LF (Code 10) alleine wird durch ein einfaches "CarriageReturn" (CR, Code 13 bzw. "\r") ersetzt. Dieses Zeichen wird innerhalb von GEOS zur Zeilentrennung verwendet.
32	CR_TO_CRLF	Jedes Auftreten eines "CarriageReturn" (CR, Code 13 bzw. "\r") wird durch ein

		"LineFeed" (LF, Code 10) **ergänzt**. Diese Kombination (CR+LF) wird in DOS-Text-Dateien als Zeilenbegrenzung verwendet.
512	CR_TO_LF	Jedes Auftreten eines "CarriageReturn" (CR, Code 13 bzw. "\r") wird durch ein "LineFeed" (LF, Code 10) **ersetzt**. Dieses Zeichen wird in Text-Dateien unter Linux und macOS als Zeilenbegrenzung verwendet.
64	DOWNCASE_CHARS	Zeichen in Kleinbuchstaben umwandeln
128	UPCASE_CHARS	Zeichen in Großbuchstaben umwandeln
256	REMOVE_HTML_TAGS	Alle "Ein-Wort-HTML-Tags" wie <b>, usw. werden entfernt. HTML-Tags, die Leerzeichen enthalten, wie <A ...>, <img ....> oder Kommentare, werden nicht entfernt. Dieses Flag wird **nach** allen anderen Operationen angewendet.

Beispiele

```
X$ = Convert$(A$, DOS_TO_GEOS)
X$ = Convert$(A$, GEOS_TO_DOS  OR  CR_to_CRLF)
X$ = Convert$(A$, WIN_TO_GEOS + DOWNCASE_CHARS, ASC("#"))
```

Alles in Großbuchstaben umwandeln:

```
X$ = Convert$(A$, UPCASE_CHARS)
```

HTML-Text in GEOS-Zeichensatz konvertieren, DOS-Zeilenumbruch (CR+LF),
 und <p> durch GEOS-Zeilenumbruch (CR) ersetzen:

```
X$ = Convert$(A$, HTML_TO_GEOS_BR + CRLF_TO_CR)
```

Hinweise:

- Die Funktionalität des Flags CRLF_TO_CR, auch einzelne LF-Zeichen zu erkennen, stellt sicher, dass alle Textdateien korrekt eingelesen werden können. Dabei ist es egal, ob sie unter DOS, Windows, macOS oder Linux erstellt wurden. Man muss die Quelle der Datei nicht kennen.
- Bei der Umwandlung zwischen Klein- und Großbuchstaben werden die lokalen Einstellungen des Computers benutzt.
- Das Ersetzen von HTML-Tags lässt sich auch vorteilhaft mit der Funktion RepalceStr$ erledigen.
- Wenn Sie einen HTML-Text haben, der ANSI (Codepage 1252, Latin-1) oder UTF-8 kodiert ist (also gültige Zeichen mit einem Code > 127 enthält) müssen sie vorher die entsprechende Konvertierungsroutine aufrufen.

```
B$ = Convert$(A$, WIN_TO_GEOS)
X$ = Convert$(B$, HTML_TO_GEOS)
```

bzw.

```
B$ = Convert$(A$, UTF8_TO_GEOS)
X$ = Convert$(B$, HTML_TO_GEOS)
```

- Convert$ verwendet für die Konvertierung zwischen GEOS und DOS- bzw. Windows-Zeichensatz eine Systemfunktion, die eine "Näherungskonvertierung" durchführt. Für nicht im neuen Zeichensatz enthaltene Zeichen wird zunächst versucht, ein "ähnlich aussehendes" Zeichen zu finden, bevor das Ersatzzeichen verwendet wird.

  Wenn Sie auf exakte Konvertierung oder weitere Codepages Wert legen, sollten Sie die Library "CodepageTools" von der R-BASIC Webseite herunterladen.

convertError

Findet Convert$ im Modus UTF8_TO_GEOS am Ende des Strings einen unvollständigen UTF-8-Code (also z.B. nur 2 von 3 erforderlichen Bytes) oder ein unvollständiges HTML-Tag (z.B. "&u") so setzt es die Systemvariable convertError auf die Anzahl der gefundenen Bytes (im Beispiel also auf 2). Andernfalls setzt es convertError auf Null.

Syntax: **<numVar>** = **convertError**

Diese Situation kann vorkommen, wenn Sie Text blockweise statt zeilenweise aus einer Datei lesen müssen, z.B. weil die Zeilen sonst zu lang sind. In diesem Fall können Sie den Dateizeiger folgendermaßen auf den Anfang des unvollständig gelesenen UTF-8-Zeichens zurücksetzen (beachten Sie das Minuszeichen!):

```
IF convertError THEN
  FileSetPos( fileVar, –convertError, TRUE )
END IF
```

Hinweise:

- Sie müssen prüfen, ob Sie sich bereits am Dateiende befinden, bevor Sie den Dateizeiger zurücksetzen. Andernfalls landen Sie in einer Endlosschleife.
- Convert$ ersetzt unvollständige UTF-8-Codes durch das Ersatzzeichen (genau ein Zeichen), unvollständige HTML-Codes werden vollständig kopiert (Anzahl = convertError). Diese Zeichen sollten Sie, wenn Sie nicht am Dateiende sind, aus dem Zielstring entfernen.
- Es wird empfohlen, für das Einlesen und konvertieren größerer Textmengen ein LargeText-Objekt zu verwenden. Dieses handelt die genannten Sonderfälle automatisch.

2.5 Programmablaufsteuerung

2.5.1 Verzweigungen

In vielen Fällen muss R-BASIC Entscheidungen treffen und je nach Situation unterschiedliche Dinge tun. Für einfache Fallunterscheidungen stehen die Befehle IF ... THEN ... ELSE (falls ... dann ... ansonsten) zur Verfügung.
Für die Unterscheidung von mehr als zwei Fällen gibt es die Anweisung ON .. SWITCH.

Entscheidungen mit IF - THEN - ELSE

Die Anweisung IF prüft den auf IF folgenden numerischen Ausdruck. Ergibt dieser einen Wert ungleich Null wird er als WAHR interpretiert und der THEN-Zweig wird abgearbeitet. Ergibt er den Wert NULL, so wird der ELSE-Zweig abgearbeitet.

Für IF stehen zwei Formate zur Verfügung: Das Standard-Format und das Kurz-Format. Beim Standard-Format muss THEN die letzte Anweisung in der Code-Zeile sein, beim Kurzformat folgen auf THEN weitere Anweisungen in der gleichen Zeile. Details dazu siehe unten.

Standard-Syntax:

IF <Bedingung> THEN
<Anweisungsfolge 1 (THEN-Zweig)>
ELSE
<Anweisungsfolge 2 (ELSE-Zweig)>
END IF

<Bedingung>
Ein numerischer Ausdruck, der WAHR (ungleich Null) oder FALSCH (gleich Null) sein kann. Dafür stehen zur Verfügung:

- einfache numerische Ausdrücke oder Variablen
- logische Operatoren (NOT, AND, OR, XOR)
- Vergleichsoperatoren (<, <=, >, >=, =, <>, =) für Zahlen und Strings. Vergleichsoperationen ergeben WAHR (Konstante TRUE, numerischer Wert: –1) wenn die Bedingung erfüllt ist oder FALSCH (Konstante FALSE, numerischer Wert Null) wenn die Bedingung nicht erfüllt ist.
- Für Variablen vom Typ FILE, HANDLE oder OBJECT und für Strukturen stehen die Vergleichsoperatoren = und <> zur Verfügung.

<THEN-Zweig>
Wird abgearbeitet, wenn die Bedingung WAHR ergibt.

<ELSE-Zweig>
Wird abgearbeitet, wenn die Bedingung FALSCH ergibt.

ELSE und <Anweisungsfolge 2 (ELSE-Zweig)> können weggelassen werden.

Beispiel:

```
IF   A > 0  THEN
       PRINT "Positiv"
ELSE
       PRINT "Nicht positiv"
END IF
```

Beispiel Standard-Format ohne ELSE-Zweig

```
IF X < 0 AND g$ = "m" THEN
  PRINT "Ich habe gar kein Auto."
END IF
```

Kurzform

In vielen Fällen müssen, falls die Bedingung wahr ist, nur einer oder wenige Befehle abgearbeitet werden. Für diesen Zweck unterstützt R-BASIC eine Kurzform der IF-Anweisung. Dabei müssen alle Anweisungen des THEN- und des ELSE- Zweigs in der gleichen Code-Zeile wie die IF-Anweisung stehen.

Syntax: **IF < Bedingung> THEN <THEN-Zweig>** : **ELSE <ELSE-Zweig>**
bzw.: **IF < Bedingung> THEN <THEN-Zweig>**

Beispiel:

```
IF Name$ = "Paul" THEN Print "Paul gefunden"
```

Hinweise:

- Hinter THEN und ELSE kann der Doppelpunkt weggelassen werden.
- in der Kurzform muss vor ELSE ein Doppelpunkt stehen.
- In der Kurzform kann END IF weggelassen werden.
- R-BASIC unterscheidet die Standard- und die Kurzform daran, ob hinter der THEN-Anweisung noch weitere Anweisungen folgen. Ist THEN nicht die letzte Anweisung in der Zeile, so wird die Kurzform aktiviert. Ein Doppelpunkt zählt dabei ebenfalls als Anweisung, Kommentare sind aber zugelassen.
- Seien Sie vorsichtig, wenn Sie Strukturvergleiche verwenden. Strukturen werden Byte für Byte verglichen, was zu Problemen führen kann, wenn sie Strings enthalten. Die vom String nicht benutzten Bytes des String-Bereichs können zufällige Werte enthalten.

Kompatibilität:

Aus Gründen der Kompatibilität werden in der Kurzform von IF auch die folgenden Formate unterstützt. Sie sollten diese Formate in eigenen Programmen aber nicht einsetzen.

Syntax: **IF < Bedingung> THEN <Zeilennummer>**
Statt THEN GOTO <Zeilennummer>.
Funktioniert nur mit Zeilennummern, nicht mit Labels.
Syntax: **IF < Bedingung> GOTO <Ziel>**
Statt THEN GOTO <Ziel>
Funktioniert sowohl mit Zeilennummern, als auch mit Labels.

Beispiele zur IF-Anweisung

Für das Normale IF-Format muss THEN am Ende einer Zeile stehen. Kommentare hinter THEN sind aber erlaubt. ELSE muss nicht allein in einer Zeile stehen. Hinter ELSE kann man den Doppelpunkt weglassen:

```
IF   A > B  THEN                ' Ich bin ein Kommentar
  A = A – 2 : B = B + 1
  ELSE  PRINT "Fertig" :  END  IF
```

Neben logischen Ausdrücken mit Zahlen können auch Zeichenketten verglichen werden (Temp$, Name$ und AlterName$ seien String-Variablen)

```
IF   Name$  > AlterName$  THEN
  Temp$ = Name$         ' Vertauschen der Namen
  Name$ = AlterName$
  AlterName$ = Temp$
END   IF
```

Logische Operatoren in IF-Anweisungen sind sehr hilfreich. Bitte beachten Sie die Hierarchie der Operatoren oder setzen Sie Klammern!

```
Label  nochmal:                  ' Rücksprungmarke
  INPUT "Bitte A eingeben"; A
  IF   A < 0   OR    A  > B  THEN
    Print "A ist ungültig"
    GOTO  nochmal
  END   IF
```

Numerische Ausdrücke sind WAHR, wenn sie nicht Null sind.

```
IF  A  THEN  Print "A ist nicht Null"
```

```
IF  17•A – B  THEN  Print "17•A  ergibt nicht B"
```

IF-Anweisungen können ineinander verschachtelt werden. Vorsicht! Das wird sehr schnell unübersichtlich und daher fehleranfällig. Die Farben deuten im Beispiel die Zugehörigkeit an.

```
IF  A > 0 THEN
  PRINT "Positiv"
ELSE
  IF  A = 0 THEN
      PRINT  "Null"
  ELSE
      PRINT  "Negativ"
  END IF
END IF
```

Mehrfachverzweigungen

Zur Unterscheidung von mehr als zwei Fällen steht die Anweisung ON ... SWITCH (je nach ... schalte um) zur Verfügung.

Syntax:

```
ON <Ausdruck> SWITCH
CASE <N1>:
    <Code>
    END CASE
CASE <N2>:
CASE <N3>:
    <Code>
    END CASE
....
DEFAULT:
    <Code>
END  SWITCH
```

Beispiel (A sei eine numerische Variable):

```
ON  A SWITCH
CASE  0:
  Print "A ist Null"
  END CASE
CASE  1:
  Print "A ist 1"
  END CASE
DEFAULT:
  Print "A nicht Null oder 1"
END  SWITCH
```

Bedeutung der Werte:

ON ... SWITCH

ON ... SWITCH (= je nach ... schalte um) leitet die Mehrfachverzweigung ein.

<Ausdruck>

<Ausdruck> ist ein numerischer Ausdruck, der einen ganzzahligen Wert (LongInt) ergeben muss.

CASE <N>

CASE (= Fall) leitet die Codesequenz ein, die abgearbeitet wird, wenn <Ausdruck> den Wert <N> ergibt.

<N> <N> ist ein fester, ganzzahliger Wert.
Einfache Berechnungen (+, -, *, /, Klammern, ^ (Exponent) , MOD (Modulo-Division), OR, AND, NOT, XOR und die Funktionen IINT(), ASC(), SQR(), FRAC(), TRUC(), SIN(), COS(), TAN(), EXP(), LN(), LOG(), LG() und SizeOf()) sowie negative Werte sind zugelassen. Variablen und sonstige Funktionen sind nicht erlaubt.

END CASE

Beendet den entsprechenden Code-Abschnitt.

DEFAULT

DEFAULT (=Vorgabe) leitet den Codeabschnitt ein, der ausgeführt wird, wenn keine der CASE <N> Bedingungen zutrifft. Der Default-Zeig ist optional.

END SWITCH

Schießt die Mehrfachverzweigung ab.

Abarbeitung von ON <Ausdruck> SWITCH... CASE

- Stößt R-BASIC auf die Anweisung ON .. SWITCH, so wird zunächst der <Ausdruck> ausgewertet.
- Daraufhin wird geprüft, welche der CASE-Bedingungen <N> zutrifft. Das geht sehr schnell, weil der Compiler einer Tabelle erstellt hat. Gegebenenfalls wird der entsprechende Code abgearbeitet.
- Stößt der R-BASIC auf die Anweisung END CASE, so wird die Mehrfachverzweigung beendet, d.h. es wird mit der auf END SWITCH folgenden Anweisung fortgesetzt.
- Gibt es zu einem CASE kein END CASE, so wird der auf das nächste CASE folgenden Code ebenfalls ausgeführt. Beispiel:

```
CASE  1:
    Print "A ist 1"
CASE  2:                    ' hier kein END CASE davor
    Print "A ist 1 oder 2"
    END CASE
```

- Findet R-BASIC keinen passenden CASE-Zweig, so wird der Code abgearbeitet, der auf die Anweisung DEFAULT folgt.
  Der DEFAULT-Zweig ist optional.

Hinweise:

- Der DEFAULT-Zweig kann weggelassen werden.
- Der Doppelpunkt hinter der CASE-Bedingung kann weggelassen werden, er dient nur der Kompatibilität.
- Direkt vor END SWITCH kann END CASE weggelassen werden.

- Hinter ON ... SWICTH und hinter End CASE kann die Zeile nicht mit einem Doppelpunkt und weiteren Befehlen fortgesetzt werden.
- Die folgende Konstruktion ist **nicht** zugelassen. IF .. THEN END CASE führt zu einem Compilerfehler:

```
  CASE  1:
    IF  B > 0  THEN END CASE
  CASE  2:
    Print "A ist 1 oder 2"
    END CASE
```

Einfache Beispiele:

```
INPUT "Geben Sie bitte A ein"; A
ON  A  SWITCH
  CASE  1:
    Print "A ist 1"
    END CASE
  CASE  2:
    Print "A ist 2"
    END CASE
  DEFAULT:
    Print "A ist weder 1 noch 2"   ' END CASE kann hier entfallen
  END SWITCH
```

Hinter CASE sind einfache Berechnungen und die Funktionen INT(), ASC() und SizeOf() zugelassen (siehe vorne).

```
DIM A$                    ' String-Variable
INPUT "Geben Sie einen Text ein"; A$
ON  ASC(A$)  SWITCH       ' ersten Buchstaben testen
  CASE  ASC("A"):
    Print "Der erste Buchstabe ist ein A"
    END CASE
  CASE  ASC("z"):
    Print "Der Text beginnt mit einem kleinen z."
                          ' END CASE kann hier entfallen
  END SWITCH
```

Beispiel für kurze CASE-Fälle

```
INPUT "Geben Sie bitte  A ein"; A
ON  A  SWITCH
  CASE  1: Print "A ist 1"  : END CASE
  CASE  2: Print "A ist 2"  : END CASE
  DEFAULT: Print "A ist weder 1 noch 2"
  END SWITCH
```

Mehrere Fälle mit dem gleichen Code:

```
INPUT "Geben Sie bitte A ein"; A
ON  A  SWITCH
  CASE  -1:
  CASE   0:
  CASE   1:
    Print "A ist -1, Null  oder 1"
    END CASE
  CASE  2: CASE 3: CASE 4:
    Print "A ist 2, 3 oder 4"
    END CASE
  CASE  -5:
  CASE   5:
    Print "A ist 5 oder -5"
    END CASE
  DEFAULT:
    Print "Nichts Passendes gefunden."
  END SWITCH
```

Verschachtelung von ON..SWITCH-Anweisungen
Die Farben deuten im Beispiel die Zugehörigkeit an. A und B sind REAL-Variablen.

```
InputBox "A und B eingeben", A,B

ON  A  SWITCH
  CASE 0:
  CASE 1:
    PRINT  "Fall 1"
    IF   B = 1  THEN   PRINT   "B ist 1"
    END  CASE
  CASE 2:
    PRINT  "Fall 2"
    ON  B  SWITCH
         CASE 1:
              PRINT  "B ist 1"
              END  CASE
         DEFAULT
              PRINT  "B ist nicht 1"
         END  SWITCH
    END  CASE
  DEFAULT
    PRINT  "sonstiges"
END  SWITCH

PRINT  "fertig."
```

Aus Kompatibilitätsgründen werden auch die folgenden Varianten unterstützt:

ON <Ausdruck> GOTO <Liste von Zielen>
ON <Ausdruck> GOSUB <Liste von Zielen>
<List von Zielen> Sprungziele, die angesprungen werden, wenn der <Ausdruck> die Werte 1, 2, 3, 4 usw. ergibt.

Beispiele (A sei eine numerische Variable)

```
ON A GOTO 10, 20, 30          ' Zeilennummern
```

```
ON A GOSUB L1, L2, L3         ' Labels
```

```
DIM A
  Input A

  ON A GOSUB L1, L2, L3
  Print "Fertig"
  End
LABEL L1
  PRINT "A ist Eins" : Return
LABEL L2
  Print "A ist Zwei" : Return
LABEL L3
  Print "A ist Drei" : Return
```

Wie Sie sehen kann man mit der GOTO bzw. GOSUB-Anweisung sehr unübersichtlichen Code erzeugen.

2.5.2 Schleifen

Sehr häufig ist es erforderlich einen bestimmen Codeabschnitt mehrfach zu durchlaufen. Ein typischer Fall ist die Ausgabe einer Tabelle oder einer Liste von Namen. Eine solche Programmstruktur bezeichnet man als Schleife. In R-BASIC stehen Ihnen drei Schleifentypen zur Verfügung:

- Die Zählschleife (auch als For-Next-Schleife bezeichnet) wird verwendet, wenn man die Anzahl der Schleifendurchläufe im Voraus kennt.
- Die WHILE - WEND - Schleife wird benutzt, wenn die Anzahl der Schleifendurchläufe nicht im Voraus bekannt ist und man die Abbruchbedingung am Schleifenanfang prüfen möchte.
- Die REPEAT - UNTIL - Schleife wird benutzt, wenn die Anzahl der Schleifendurchläufe nicht im Voraus bekannt ist und man die Abbruchbedingung am Schleifenende prüfen möchte.
- Mit den Schlüsselworten CONTINUE und BREAK kann man Schleifendurchläufe vorzeitig beenden oder die Schleife vorzeitig verlassen.
- Verwenden Sie niemals GOTO um eine Schleife zu verlassen! Schleifen erzeugen einen Eintrag auf dem Return-Stack. GOTO räumt diesen Eintrag nicht auf und der Stack kann überlaufen.
- Eine Return-Anweisung innerhalb einer Schleife ist erlaubt. Return räumt den Stack sauber auf.

Zähl-Schleifen

Die Zählschleife mit FOR wird verwendet, wenn man die Anzahl der Schleifendurchläufe im Voraus kennt. Die FOR-Schleife wird immer mindestens einmal durchlaufen.

Syntax: **FOR** **<N>** = **startwert** **TO** **endwert** [**STEP** **schrittweite**]
< Anweisungen >
NEXT **<N>**

<N>: numerische Variable, "Schleifenzähler"
Jeder numerische Datentyp ist zulässig (auch WWFixed). Aus historischen Gründen wird für die Zählvariable häufig N, I oder K verwendet.
startwert: Anfangswert für den Schleifenzähler
Der Schleifenzähler wird am Beginn mit diesem Wert belegt.
endwert: Endwert für den Schleifenzähler
Die Schleife wird verlassen wenn <N> den Endwert erreicht oder überschritten hat.
schrittweite: (optional) Dieser Wert bestimmt, um welchen Wert der Schleifenzähler nach jedem Durchlauf erhöht wird. Negative Werte sind zulässig, dann wird der Schleifenzähler vermindert. Der Standardwert für schrittweite ist 1.

Das folgende Beispiel gibt die Zahlen von 1 bis 4 aus:

```
DIM N
  FOR N = 1 TO 4
    Print N
    NEXT N
```

Bedeutung der einzelnen Elemente:

FOR Die Anweisung "FOR N = startwert" (Für N = ..) eröffnet die Schleife und belegt den Schleifenzähler N mit dem Startwert.

TO Die Anweisung "TO endwert" (bis ...) legt den Endwert des Schleifenzählers fest.

STEP Die optionale Anweisung "STEP schritteweite" (Stufe ..) legt die Schrittweite fest. Sie wird nach jedem Durchlauf auf den Schleifenzähler addiert. Negative Schrittweiten sind zulässig. Der Standardwert (ohne STEP) ist 1.

Es wird empfohlen, dass Startwert, endwert und Schrittweite jeweils ganzzahlig sind.

NEXT Die Anweisung NEXT N (Nächstes N) schließt die Schleife. Der Schleifenzähler N wird erhöht (bei negativer Schrittweite vermindert) und die Abbruchbedingung wird geprüft. Die Angabe des Schleifenzählers ist optional (Abwärtskompatibilität).

NEXT bricht die Schleife ab, wenn der Wert "aktuelle Schleifenzähler + Schrittweite" größer als der Endwert (bei negativer Schrittweite: kleiner als der Endwert) ist. Ansonsten - also auch wenn der Endwert genau erreicht ist - wird der neue Wert dem Schleifenzähler zugewiesen und die Schleife wird erneut durchlaufen.

Beispiele:

```
FOR  N = 0  TO 10                ' 11 Durchläufe
  PRINT N, N•N
NEXT N
```

```
FOR  N = 1  TO 10  STEP  0.5     ' 20 Durchläufe
  PRINT N, N•N
NEXT  N
```

Schleifen können verschachtelt werden. Die Farben deuten die Zugehörigkeit an.

```
FOR X = 10 TO 30
  FOR Y = 24 TO 58
      PSet X, Y
  NEXT  Y
NEXT  X
```

Hinweise:

- Der Schleifenrumpf wird in jedem Fall einmal durchlaufen, da die Abbruchbedingung erst am Ende der Schleife (von NEXT) geprüft wird.

- Eine Schleifendurchlauf kann mit BREAK oder CONTINUE vorzeitig abgebrochen werden.
- Der Schleifenzähler darf im Schleifenrumpf verändert werden, z.B. durch Zuweisung eines Wertes weit über dem Endwert. Auch dadurch kann man einen vorzeitigen Schleifenabbruch erzwingen.
- Ein Unterprogramm darf aus einer Schleife heraus mit RETURN verlassen werden.
- Startwert, Endwert und Schrittweite sollten ganzzahlig sein, sonst kann es zu unerwarteten Problemen kommen. Durch die interne Zahlendarstellung kann es zu Rundungsfehlen beim ständigen Aufaddieren der Schrittweite kommen, so dass der Endwert nicht exakt erreicht wird. Im folgenden Beispiel führt die minimale Abweichung von 4.33E-19 dazu, dass die Schleife **NICHT** mit dem Wert 4 durchlaufen wird.

```
FOR  N = 0  TO 4  STEP 0.4
  PRINT N, N•N
NEXT  N
PRINT  N - 4        ' ist NICHT Null!
```

  In vielen Fällen kann man dieses Problem umgehen, indem man als Endwert den gewünschten Wert + halbe Schrittweite (hier also 4.2) angibt.

  FOR N = 0 TO 4.2 STEP 0.4

Aus Gründen der Abwärtskompatibilität sind die folgenden Syntaxvarianten für NEXT erlaubt:

- Die Angabe des Schleifenzählers hinter Next ist optional. NEXT ohne Angabe eines Schleifenzählers schließt genau eine Schleife. Vorsicht also bei verschachtelten Schleifen.

```
FOR  N = 0  TO 4
  PRINT N, N•N
NEXT
```

- Mehrere Schleifenzähler hinter NEXT sind zulässig. Der Zähler der inneren Schleife muss zuerst angegeben werden.

```
FOR X = 10 TO 30
  FOR Y = 24 TO 58
      PSet X, Y
  NEXT  Y, X
```

Tipp:

- Die Variante mit NEXT ohne Schleifenzähler läuft merklich schneller, weil R-BASIC keine Prüfung ausführen kann, ob der Schleifenzähler korrekt ist. Andererseits werden eventuelle Fehler in der Programmstruktur schwerer erkannt.

Die WHILE - WEND -Schleife

Die While-Wend-Schleife wird verwendet, wenn die Anzahl der Schleifendurchläufe noch nicht im Voraus feststeht. While-Wend ist abweisend, d.h. ist die Bedingung hinter While von Anfang an FALSCH, wir die Schleife nie durchlaufen.

Syntax: **WHILE** **<Bedingung>**
< Anweisungen >
WEND

Syntax einzeilig (Doppelpunkt hinter der Bedingung beachten):
WHILE **<Bedingung>** : **< Anweisungen >** : **WEND**

<Bedingung>: Numerischer Ausdruck. Die Schleife wird durchlaufen, wenn <Bedingung> einen Wert ungleich Null ergibt.

Beispiel

```
DIM A
  Print "Geben Sie eine Zahl größer als Null ein."
  A = –1
  WHILE A <= 0
    Input "Positive Zahl"; A
  WEND
  Print "Prima."
```

Bedeutung der einzelnen Elemente:

WHILE <Bedingung>

Die Anweisung WHILE (Solange) eröffnet die Schleife. Die auf WHILE folgende Bedingung (ein numerischer Ausdruck) wird geprüft. Ergibt sie den Wert wahr (also ungleich Null) wird der Schleifenrumpf durchlaufen. Ergibt die Bedingung den Wert falsch (gleich Null) wird die Schleife verlassen. Es wird mit der auf Wend folgenden Anweisung fortgesetzt.

Die Schleife wird durchlaufen, solange die Bedingung erfüllt ist.

WEND Die Anweisung WEND (Wende) schließt die Schleife, indem zur WHILE-Anweisung zurück gesprungen wird.

END WHILE kann anstelle von **WEND** verwendet werden. Funktionell besteht kein Unterschied.

Hinweise:

- Der Schleifenrumpf <Anweisungen> wird solange durchlaufen, wie die Bedingung wahr ist.
- Ist die Bedingung von Anfang an falsch, wird der Schleifenrumpf nie durchlaufen.
- Zu jedem WHILE muss es genau ein WEND geben und umgekehrt. WHILE muss im Programmcode immer vor WEND stehen.

- R-BASIC interpretiert jeden numerischen Ausdruck, der nicht Null ergibt, als "wahr". Ergibt er Null, ist er "falsch" (FALSE).
- Die Bedingung darf Verknüpfungen mit logischen Operatoren enthalten.
- Ein Unterprogramm darf aus einer Schleife heraus mit RETURN verlassen werden.
- Eine Schleifendurchlauf kann mit BREAK oder CONTINUE vorzeitig abgebrochen werden.

Beispiele:

```
A =  12
B = 34
WHILE   A > 0   AND  B > 0
  A = A – 1
  B = B – 10
  PRINT A, B
WEND
```

```
' einen Text verlängern
' Aus "Hallo Welt" wird "Hallo Welt.........."
InputBox "Bitte einen Text eingeben"; C$
WHILE   Len(C$) < 20
  C$  =  C$ + "."
WEND
PRINT  C$ + "!"
```

```
' warten auf die Enter-Taste
WHILE  InKey$ <> Chr$(13) : WEND     ' Doppelpunkt beachten
```

Die REPEAT - UNTIL - Schleife

Die Repeat-Until-Schleife wird verwendet, wenn die Anzahl der Schleifendurchläufe noch nicht im Voraus feststeht. Repeat-Until ist nicht abweisend, d.h. die Schleife wird in jedem Fall mindestens einmal durchlaufen.

Syntax: **REPEAT**
< Anweisungen >
UNTIL <Bedingung>

Syntax einzeilig (Doppelpunkt hinter der Bedingung beachten):
REPEAT < Anweisungen > : **UNTIL <Bedingung>**

<Bedingung>: Numerischer Ausdruck. Die Schleife wird erneut durchlaufen, solange <Bedingung> den Wert Null ergibt.

Bedeutung der einzelnen Elemente:

REPEAT
Die Anweisung REPEAT (Wiederhole) eröffnet die Schleife.

UNTIL <Bedingung>
Die Anweisung UNTIL (bis ...) schließt die Schleife. Die auf UNTIL folgende Bedingung (ein numerischer Ausdruck) wird geprüft. Ergibt sie den Wert wahr (also ungleich Null) wird die Schleife verlassen. Es wird mit der auf UNTIL folgenden Anweisung fortgesetzt. Ergibt die Bedingung den Wert falsch (gleich Null) wird der Schleifenrumpf erneut durchlaufen.
Die Schleife wird solange durchlaufen bis die Bedingung erfüllt ist.

Hinweise:

- Der Schleifenrumpf <Anweisungen> wird in jedem Fall mindestens einmal durchlaufen.
- Der Schleifenrumpf wird solange durchlaufen, bis die Bedingung wahr ist.
- R-BASIC interpretiert jeden numerischen Ausdruck, der nicht Null ergibt, als "wahr". Ergibt er Null, ist er "falsch" (FALSE).
- Die Bedingung darf Verknüpfungen mit logischen Operatoren enthalten.
- Zu jedem REPEAT muss es genau ein UNTIL geben und umgekehrt. REPEAT muss im Programmcode immer vor UNTIL stehen.
- Ein Unterprogramm darf aus einer Schleife heraus mit RETURN verlassen werden.
- Eine Schleifendurchlauf kann mit BREAK oder CONTINUE vorzeitig abgebrochen werden.

Beispiele:

```
A = -7
REPEAT
  InputBox "Geben Sie eine positive Zahl ein", A
UNTIL   A > 0
PRINT  A            ' A ist jetzt immer positiv
```

```
A =  12
B = 34
REPEAT
  A = A – 1
  B = B – 10
  PRINT   A, B
UNTIL  A <= 0 AND  B <= 0
```

```
' einen Text verlängern
' Aus "Hallo Welt" wird "Hallo Welt........."
InputBox "Bitte einen Text eingeben", C$
REPEAT
  C$  =  C$ + "."
UNTIL    Len(C$)  >=  20
PRINT  ">>" + C$ + "<<"
```

```
' warten auf die Enter-Taste
REPEAT UNTIL InKey$ = Chr$(13)  ' Doppelpunkt ist nicht nötig
```

```
! Schleifenabbruch mit Enter
REPEAT
  x = 1000* RND()               ' Zufallszahl von 0 bis 999
  Print  x
UNTIL    InKey$ = Chr$(13)
Print "Abbruch erfolgte bei"; x
```

BREAK und CONTINUE

Break und Continue beenden eine Schleifendurchlauf oder eine Schleife vorzeitig.

Syntax: **BREAK**
Syntax: **CONTINUE**

BREAK beendet eine Schleife (While-Wend, For-To-Next oder Repeat-Until) vorzeitig. Die Abarbeitung wird mit dem auf die Schleife folgenden Befehl (also hinter Wend, Next oder Until) fortgesetzt.

CONTINUE beendet einen Schleifendurchlauf einer While-Wend, For-To-Next oder Repeat-Until Schleife vorzeitig. Die Schleifenbedingung wird erneut geprüft, da CONTINUE zum Schleifen-End-Befehl (Wend, Next oder Until) springt.

Wichtig: Verwenden Sie niemals GOTO um eine Schleife zu verlassen! Schleifen erzeugen einen Eintrag auf dem Return-Stack. GOTO räumt diesen Eintrag nicht auf und der Stack kann überlaufen. Eine Return-Anweisung innerhalb einer Schleife ist hingegen erlaubt. Return räumt den Stack sauber auf.

Beispiele:

```
! Auslassen von bestimmten Zahlen in einer Schleife
FOR  N = 1  TO  5
  IF  N = 3 THEN  CONTINUE
  Print N
NEXT
```

```
Ausgabe:  1
          2
          4
          5
```

```
! Vorzeitiger Abbruch einer Schleife
FOR  N = 1  TO  5
  IF  N = 3 THEN  BREAK
  Print N
NEXT
```

```
Ausgabe:  1
          2
```

```
! Schleifenabbruch per Zufall
REPEAT
  x = 1000* RND()                ' Zufallszahl von 0 bis 999
  IF  x > 990  THEN  BREAK
  Print  x
UNTIL  FALSE                     ' Endlos-Schleife
Print "Abbruch erfolgte bei"; x
```

2.5.3 Pause und Delay

Die Befehle Pause und Delay dienen dazu, den Programmablauf für eine bestimmte Zeit anzuhalten. Pause und Delay werden vor allem in **klassischen** BASIC-Programmen verwendet. In objektorientierten Programmen sind Pause und Delay zu Fehlersuche hilfreich. Ansonsten sollten Sie diese Befehle in objektorientierten Programmen nicht verwenden. Stattdessen sollten Sie einen Timer verwenden, wenn Sie einen zeitgesteuerten Programmablauf wünschen. Timer sind ausführlich im Handbuch "Spezielle Themen", Kapitel 16, beschrieben.

PAUSE

Der Pause-Befehl bewirkt eine kurze Programmunterbrechung.

Syntax: **PAUSE** [**n**]
n: num. Wert, Dauer in 0,1 Sekunden-Schritten
Defaultwert ohne n: 0,1 Sekunde Pause

Beispiel:

```
PAUSE 2.5       ' Eine viertel Sekunde warten
```

DELAY

Programm-Verzögerung. Der Befehl DELAY (verzögere) wartet, bis seit dem letzten DELAY-Befehl eine bestimmte Zeit vergangen ist. Im Gegensatz zum PAUSE-Befehl werden die zwischenzeitlich abgearbeiteten Befehle berücksichtigt. Damit ist eine gezielte Steuerung der Programmablaufgeschwindigkeit möglich. Nach Möglichkeit sollen Sie, z.B. für Spiele, statt Delay einen Timer verwenden.

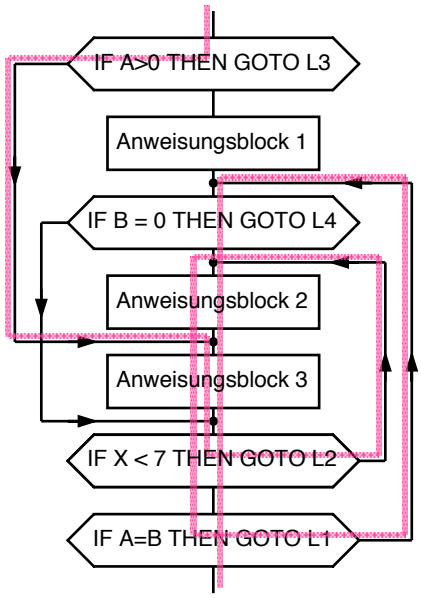
Syntax: **DELAY** [**InitWert**]
InitWert: numerischer Wert. InitWert bestimmt die Timer-Tics (1/60 s), die DELAY mindestens warten soll. DELAY 0 schaltet die Verzögerung ab.

Das Programm wird erst fortgesetzt, wenn seit dem letzten DELAY-Befehl (mit oder ohne InitWert) mindestens "InitWert" Timer-Tics vergangen sind. Ist bereits eine längere Zeit vergangen wird das Programm sofort fortgesetzt.
Achtung! Wird ein initWert angegeben, so wartet der DELAY-Befehl nicht, sondern stellt nur die Wartezeit für den nächsten DELAY-Befehl ein.

Beispiel: Langsame Ausgabe eines Textes, Buchstabe für Buchstabe

```
DELAY 30                    ' Eine halbe Sekunde
Input A$
FOR N = 1 To Len(a$):
  PRINT Mid$(a$, N, 1);     ' Einen Buchstaben ausgeben
  DELAY
NEXT
```

2.5.4 Unbedingte Sprünge



Schleifen und Verzweigungen springen zu einem anderen Programmteil, wenn bestimmte Bedingungen erfüllt sind. Ein unbedingter Sprung hingegen wird in jedem Fall ausgeführt. Sie sollten die häufige Verwendung des GOTO-Befehls vermeiden, da er schnell zu sehr unübersichtlichen Programmen (dem sogenannten Spaghetti-Code, siehe Bild) führen kann.

GOTO

Die Anweisung GOTO (Gehe zu) setzt den Programmablauf an der angegebenen Stelle fort.

Syntax: **GOTO** <**sprungZiel**>

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

LABEL

Die Anweisung LABEL (Marke) vereinbart ein Ansprungziel für GOTO, GOSUB oder RESTORE. Siehe auch Kapitel 2.11.4 (Data-Zeilen).

Beispiel:

```
  IF A > 0 THEN GOTO MeineMarke    ' Springt nach unten
  <Anweisungen 1>     ' Werden ausgeführt wenn A nicht > 0 ist
LABEL  MeineMarke:
  <Anweisungen 2>     ' Werden in jedem Fall ausgeführt
```

In den meisten Fällen kann man die Verwendung des GOTO-Befehls vermeiden, wenn man das Programm anders strukturiert. Der folgende Code ist identisch mit dem Beispiel oben. Durch das Umkehren der Bedingung in der IF-Anweisung (IF A <= 0 statt IF A > 0) wird weder ein GOTO-Befehl noch Label benötigt. Zusätzlich wird das Programm übersichtlicher.

```
  IF A <= 0 THEN
    <Anweisungen 1>   ' Werden ausgeführt wenn A nicht > 0 ist
    End IF
  <Anweisungen 2>     ' Werden in jedem Fall ausgeführt
```

Im Allgemeinen kann man das "umspringen" von Code mit GOTO durch eine IF-Anweisung ersetzen, wobei die Sprungbedingung negiert werden muss (d.h. aus = wird <>, aus < wird >= usw.). "Rückwärtssprünge" mit GOTO lassen sich meist durch eine REPEAT-UNTIL-Schleife ersetzen, wobei die Bedingung ebenfalls negiert werden muss.

Beispiel

```
' Code mit GOTO
LABEL marke1
  INPUT "Geben sie eine positive Zahl ein"; A
  IF A <= 0 THEN GOTO marke1

' Code mit einer Schleife
REPEAT
  INPUT "Geben sie eine positive Zahl ein"; A
  UNTIL A > 0
```

Abwärtskompatibilität

R-BASIC unterstützt auch die in vielen BASIC-Dialekten verwendete Kombination "GOTO Zeilennummer". Das kann die Übertragung fremder BASIC-Programme vereinfachen. Die "Zeilennummer" muss dabei explizit angegeben sein (z.B. "1000 CLS ...", siehe Beispiel). Sie sollten diese Variante in eigenen Programmen nicht verwenden.

Beispiel:

```
    GOTO 1000     ' verzweigt das Programm nach unten
    ....          ' Dieser Teil wird übersprungen
1000 CLS
    ...           ' hier geht es dann weiter
```

2.5.5 Vorzeitiger Programmabbruch

In den meisten Fällen wird der Nutzer das Programm regulär über das Dateimenü beenden. R-BASIC bietet jedoch auch die Möglichkeit, den Programmablauf vorzeitig per BASIC-Befehl zu beenden.

EXIT

EXIT bricht die Programmausführung ab und schließt das Programm. Die Wirkung ist die gleiche als ob der Nutzer den Menüeintrag "Beenden" (oder "Verlassen") aus dem Dateimenü gewählt hat. EXIT kann an beliebiger Stelle im Programm stehen, auch in Schleifen und innerhalb von Unterprogrammen oder Action-Handlern.

Syntax: **EXIT**

Beispiel:

```
IF  X < 0  THEN  EXIT
```

END

END ist ein Befehl zur Wahrung der Abwärtskompatibilität zu anderen BASIC-Dialekten. In R-BASIC sollten Sie END nicht verwenden.

END bricht den laufenden Programmteil ab, das Programm wird jedoch nicht geschlossen. END ist in klassischen BASIC-Programmen der übliche Weg, das Programm vorzeitig zu beenden. In R-BASIC bleibt ein Programm nach einem END-Befehl weiterhin funktionsfähig. END kann an beliebiger Stelle im Programm stehen, auch in Schleifen und innerhalb von Unterprogrammen oder Action-Handlern.

Syntax: **END**

Verwechseln Sie END nicht mit RETURN (siehe Kapitel 2.6, Unterprogramme). RETURN bewirkt, dass das Unterprogramm zurückkehrt, der auf den Aufruf des Unterprogramms (Sub oder Function) folgende Code wird abgearbeitet. END hingegen würgt den laufenden Handler komplett ab.

Beispiel:

```
 IF  X > 0  THEN  END
```

(Leerseite)

(Leerseite)