

```

*****
*****
**
**      Das QBasic Kochbuch V2.1
**      =====
**      von Thomas Antoni, 7. 3. 1999 - 04. 03. 2006
**
**      Eine Bauanleitung für QBasic-Programme mit Hinweisen auf die
**      Unterschiede zu MS QuickBASIC und PowerBASIC. Das Kochbuch
**      steht inklusive der 48 Beispielprogramme auf www.qbasic.de in
**      der Tutorial-Rubrik zum Download bereit.
**
*****

```

```

*****
* Inhalt
*****

```

```

Seite
-----
2 Vorwort
2 Online-Hilfe verwenden
3 Bedienung und Aufruf von Editor, Interpreter, Compiler und EXE-Programmen
4 Syntax
4 Datentypen und Variablen
6 Felder
8 Konstanten (CONST, DATA, READ)
8 Mathematische Funktionen, Operatoren, Wertzuweisungen
10 Textverarbeitung - Manipulation von Zeichenketten (Strings)
11 Textanzeige, Farben
13 Tastatureingaben
15 Grafiken anzeigen
19 Sound aus PC-Speaker ausgeben
20 Joystickabfrage
20 Wartezeiten erzeugen und Datum/ Uhrzeit bearbeiten
21 Zufallszahlen erzeugen
21 Schleifen und Verzweigungen
24 Allgemeines zu Subroutinen und Funktionen (Parameter, Lokal-/Globalvariablen)
26 Subroutinen (Unterprogramme)
27 Funktionen (Unterprogramme mit Rückgabewert)
28 Lokale Subroutinen (GOSUB)
28 Lokale Funktionen (DEF FN...)
29 DOS-Befehl oder externes EXE-Programm/ BAT-Batchdatei aufrufen
29 Modulare Programmierung und Bibliotheken (CHAIN, *.LIB)
30 Dateibearbeitung - Allgemeines, Dateiarten, Fehlerbehandlung
33 Sequentielle Dateien
35 Direktzugriffs-Dateien mit TYPE-Puffer
36 Direktzugriffs-Dateien mit FIELD-Puffer
38 Binäre Dateien
39 DruckerAusgabe
39 Serielle Schnittstellen
40 Direkter Speicherzugriff und I/O-Port-Zugriff
43 Umstieg von QBasic nach MS QuickBASIC
43 Umstieg von QBasic nach PowerBASIC
44 Tipps zu häufig vorkommenden Programmierproblemen
46 Internet-Links zu QBasic
47 Literatur zu QBasic
48 Liste der Beispielprogramme

```

```

*****
* Vorwort
*****
Vorwort zur Kochbuch-Version 1.0, 21.12.1999
-----

```

Dieses QBasic-Kochbuch ist eine 'Bauanleitung' für QBasic-Programme. Für alle wichtigen Programmieraufgaben werden in übersichtlicher Form die benötigten QBasic-Befehle genannt und erläutert - illustriert durch eine Fülle von Programmbeispielen. Das QBasic-Kochbuch ist gleichzeitig eine Kurzreferenz fast aller QBasic-Befehle und soll die sehr gute Online-Hilfe von QBasic 1.1 ergänzen.

Das QBasic-Kochbuch ersetzt jedoch nicht einen Einführungskurs in QBasic; Neueinsteigern seien hierfür die hervorragenden Tutorials empfohlen, die, ebenso wie das vorliegende Kochbuch, auf meiner Webseite [www.qbasic.de](http://www.qbasic.de) in der Rubrik "QBasic -> Tutorials" zum Herunterladen bereitstehen. Auf [www.qbasic.de](http://www.qbasic.de) finden Sie auch viele weiterführende Informationen für fortgeschrittene Programmierer, eine Reihe von kostenlos downloadbaren elektronischen Büchern (E-Books) und die gigantische QB-MonsterFAQ, die fast alle Fragen beantwortet, die jemals in den diversen Foren zu QBasic gestellt wurden.

QBasic ist ein reiner Interpreter und kostenlose Beigabe von MS-DOS ab V5.0 sowie Windows 95/98. Umsteiger auf die echten Compiler QuickBASIC und PowerBASIC (ehemals Borland TurboBASIC), mit denen man eigenständige EXE-Programme erstellen kann, erhalten die wichtigsten Informationen für den Umstieg.

Die Literaturhinweise beziehen sich auf die am Schluss aufgelisteten Bücher und sind wie folgt aufgebaut: {x/n} = Buch {x}, Seite Nummer n.

Credits: Dank an Hawkynt für seine Ergänzungen und Korrekturen zum Kapitel "Umstieg von QBasic nach PowerBASIC V3.5"

```

Vorwort zur Kochbuch-Version 2.0, 10.10.2005
-----

```

Die unerwartet große Resonanz auf die erste Auflage meines QBasic-Kochbuchs hat mich motiviert, ständig an dem Kochbuch weiterzuarbeiten. In der neuen Version 2.0 habe ich nun zahlreiche Korrekturen und Erweiterungen eingearbeitet, die sich auch aufgrund von vielen Leserbriefen in den Jahren 1999 bis 2005 angesammelt haben. Vielen Dank an alle, die mir Anregungen mailten. Alle 48 Beispielprogramme liegen der Download-Version des Kochbuchs jetzt bei und sind im letzten Kapitel aufgelistet. Das QBasic-Kochbuch steht in einer HTML-Version und in einer PDF-Version zur Verfügung. Die PDF-Version ist ideal zum Ausdrucken geeignet. In der HTML-Version können Sie die Beispielprogramme ganz bequem durch Anklicken öffnen und starten.

Für die Syntax-Beschreibungen der einzelnen Befehle verwendet das Kochbuch die im Kapitel 'Syntax' erläuterten Symbole.

```

*****

```

- ```

* Online-Hilfe verwenden
*****

```
- Ein rechter Mausklick in der QBasic-Entwicklungsumgebung auf ein Schlüsselwort im Programmlisting oder im Hilfefenster öffnet die Hilfe zum Befehls-Schlüsselwort. Alternativ kann die F1-Taste verwendet werden.
  - Bei PowerBASIC ist die kontextsensitive Hilfe zu dem Schlüsselwort, auf dem der Cursor steht, über <Strg + F1> erreichbar.
  - Im Hilfefenster kann mit <Bearbeiten> <Suchen> über alle Hilfethemen hinweg nach einem Begriff gesucht werden, zu dem man mehr wissen will (sehr nützlich!).
  - Hilfe beenden mit Esc-Taste

```

*****
* Bedienung und Aufruf von Editor, Interpreter, Compiler und EXE-Programmen
* {11/25+456}
*****
- Programm während Interpretation abbrechen: [Strg+Pause] (auch bei Absturz),
  während Anwendereingaben über den INPUT-Befehl erfolgt der Programm-
  Abbruch mit [Strg+C] )
- Direktstart eines Programms *.BAS von DOS aus:
  QBASIC /RUN *.BAS 'Start eines QBasic-Programms ohne die Entwick-
  lungsumgebung zu öffnen. Das Programm kehrt zu
  'DOS zurück, wenn es mit SYSTEM statt END
  'abgeschlossen ist
- QBASIC /H - startet QBasic mit der vollen VGA-Auflösung von 50 Zeilen statt
  mit 25 Zeilen (QB /H bei QuickBASIC).
- QBASIC /EDITOR [/H] <Filename> - startet den in QBASIC enthaltenen MS-DOS-
  Texteditor [im 50-zeiligen VGA-Modus]; Texteditor nur in QBasic (max.
  Dateilänge 64 K), nicht in QuickBASIC vorhanden!
- Debug-Funktionen (d.h. Funktionen zur Fehlersuche; unter Windows funktionieren
  einige Funktionen u.U. nicht): {11/459}
  - Haltepunkte setzen/ rücksetzen: <Cursor auf den gewünschten Befehl set-
  zen>, dann <Debug|Haltepunkt ein/aus> oder [F9]
  - Am Haltepunkt Variablen anschauen: Ist im "Direkt"-Fenster durch die
  Eingabe von PRINT <Variable> möglich. Ausgabebildschirm über
  <Ansicht | Ausgabebildschirm> oder [F4]-Taste kontrollierbar.
  Bei QuickBASIC u. PowerBASIC ist die Variablenanzeige komfortabler
  möglich über <Debug|Variable anzeigen> bzw. <Break|Watch|Add
  Watch>
  - Programme in Zeitlupe ablaufen lassen mit Hervorhebung des jeweils ak-
  tuellen Befehls und kurzem Halt an den Schleifenanfängen:
  <Debug|Rückverfolgung ein> <Ausführen|Start> oder den fragli-
  chen Programmbereich durch die Befehle TRON und TROFF einklammern
  (Trace On/Off). Die Zeitlupe ist nicht immer praktikabel, besonders
  im Windows-DOS-Fenster.
  - Vom aktuellen Befehl aus im Einzelschritt Befehl für Befehl ausführen:
  <Debug|Einzelschritt> oder [F8]. Sollen SUBS und FUNCTIONS normal
  (ohne Einzelschritt) durchlaufen werden, so erfolgt die Schrittfort-
  schaltung über <Debug|Prozedurschritt> oder [F10]. Bei jeder
  Betätigung der [F8]- oder [F10]-Taste wird ein Befehl abgearbeitet
  und im Quellcode hell dargestellt.
  - Aktuellen Befehl festlegen - zum Überspringen eines Programmabschnitts:
  <Cursor auf den gewünschten Befehl setzen> <Debug | nächste Anwei-
  sung festlegen>
- Anzeige von Unterprogrammen mit [F2]
  Durchblättern der Unterprogramme mit [Shift+F2], rückwärts mit [Strg+F2].
- Ein zweites Editierfenster kann mit <Ansicht|Aufteilen> geöffnet werden, z.B.
  um Hauptprogramm und Subroutine gemeinsam auf dem Bildschirm zu haben. {6/88}
- Max. Länge von Code und Daten: Insgesamt 160K (bei QuickBASIC und PowerBASIC
  nur durch den freien Speicherplatz im konventionellen unteren 640K-Speicher
  begrenzt). Das Hauptprogramm und die einzelnen Unterprogramme (SUBS und
  FUNCTIONS) dürfen im Quellcode jeweils maximal 64 KB lang sein.
- Library-Funktion (nur bei QuickBASIC und PowerBASIC): Über 'QB /L' lässt sich
  QuickBASIC mit der Qick-Library QB.QLB aufrufen: (erforderlich z.B. bei Ver-
  wendung der Befehle INTERRUPT[X], CALL ABSOLUTE ...) {9/6}. Der Program-
  mierer kann auch eigene Bibliotheken erstellen, in denen häufig benötigte
  Routinen (SUBS und FUNCTIONS) zusammengefasst sind.
- Bei QuickBASIC lassen sich die Pfade zu den Include-, Lib- usw. -Files indivi-
  duell setzen {9/2}
- Parameterübergabe von der Aufruf-Kommandozeile an ein kompiliertes Quick-
  Basic-Programm: Das Programm kann die übergebenen Aufrufparameter über
  COMMANDS abfragen (nur von QuickBASIC, nicht von QBasic unterstützt).
  Beispiele: - PRINT COMMANDS 'Zeigt den übergebenen Text-Parameter an
  - a% = VAL(COMMANDS): PRINT a% 'Zeigt einen übergebenen
  'Zahlenwert-Parameter an
Zum Testen gibt man in der Entwicklungsumgebung vor dem Starten des Programms
wie folgt einen Parameter vor:
<Ausführen | Ändere COMMANDS... | <Parameterwert> >
- Der Compiler (zum Erzeugen von EXE- und OBJ-Dateien) und der Linker (zum
  Einbinden externer Programmmodule und Bibliotheken) sind bei QuickBASIC und
  PowerBASIC auch als eigenständige EXE-Programme vorhanden, um mehr Arbeits-
  speicher für die Bearbeitung großer Programme zur Verfügung zu stellen.

```

```

*****
* Syntax
*****
Die Syntax ist quasi die Vorschrift für die korrekte Rechtschreibung und
Grammatik der QBasic-Befehle. Dieses Kochbuch verwendet Großbuchstaben für die
Befehlsschlüsselwörter sowie die folgenden Symbole für die Syntaxbeschreibungen
der einzelnen Befehle.
- [] = Der Ausdruck innerhalb der eckigen Klammern ist optional und muss
  nicht zwingend angegeben werden. Die Klammern sind nicht mit
  einzugeben
- < > = Zwischen den spitzen Klammern steht die Kurzbezeichnung eines
  Syntaxelements (z.B. "Variable" oder "Tastencode"), der bei dem tat-
  sächlichen Befehl durch die konkreten Angabe (z.B. Variablenname oder
  Tasten-Codenummer) zu ersetzen ist. Die Klammern werden im Quelltext
  nicht mit angegeben.
- {} = In der geschweiften Klammer stehen Alternativen, von denen nur eine
  verwendet werden darf. Die Alternativen sind durch senkrechte Striche
  voneinander getrennt. Die Syntaxangabe "M[B|F]" bedeutet z.B. dass
  sowohl "MB" als auch "MF" erlaubt sind.
Die Syntaxbeschreibungen der einzelnen Befehle sind im Kochbuch durch rote
Schrift hervorgehoben.
Generell gelten bei QBasic die folgenden Syntaxregeln:
- 1 Befehl darf nicht länger als 1 Zeile sein; max. Zeilenlänge 256 Zeichen
- In eine Zeile darf man mehrere Befehle schreiben, jeweils durch ':'
  voneinander getrennt. Beispiel: CLS: LOCATE 12,35: PRINT "Hallo"
- Kommentar kann man hinter ein Hochkomma ' oder REM einfügen (REM nur am
  Befehls-Anfang). Kommentar dient zur Erläuterung des Programms und wird bei
  Programmausführung überlesen.
- Zeilennummern sind nicht erforderlich, können aber optional verwendet werden.
- Sprungmarken: <Zahl> oder <Name>;, z.B. '120' oder 'Start:'
- Ein Programm wird beendet mit dem Befehl 'END' oder 'SYSTEM'. SYSTEM bewirkt
  Aussprung zu DOS aus dem Interpreter heraus, wenn das Programm vom außer-
  halb des QBasic-Interpreters mit 'QBASIC /RUN prog.BAS' aufgerufen wurde. END
  bewirkt einen Rücksprung zur QBasic-Entwicklungsumgebung.
- Namen von Variablen, Subroutinen und Funktionen: Max. 40 Zeichen, nur Buch-
  staben, Ziffern und Punkt, 1. Zeichen=Buchstabe (Beispiel: 'Parfum 4711').
  Ein Unterstrich "_" ist nicht erlaubt. Groß-/Kleinschreibung wird nicht unter-
  schieden. Befehlsschlüsselwörter, z.B. 'print', dürfen nicht verwendet werden.
*****
* Datentypen und Variablen
*****
- Datentypen werden im Variablennamen durch ein nachfolgendes Typkennzeichen
  (Suffix) gekennzeichnet. Variablen-Deklaration sind nicht erforderlich, aber
  mit DIM möglich; s.u. QBasic kennt folgende Datentypen (am Anfang ist hier
  jeweils ein beispielhafter Variablenname mit Typkennzeichen aufgeführt):
  Variable| Datentyp| Erläuterung, Wertebereich
  -----+-----
- anna% : INTEGER, Ganzzahl mit Vorzeichen (16 bit) -32768...32767
- otto& : LONG, lange Ganzzahl mit Vorzeichen (32 bit)
  -2147483648...2147483647
- egon! : SINGLE, einfach lange Gleitpunktzahl (32 Bit, 7 Stellen genau)
  +2,802597 *10^-45...+3,402823 *10^38
- egon : dito (der Suffix '!' ist bei SINGLE-Variablen weglassbar)
- paul# : DOUBLE, doppelt lange Gleitpunktzahl (64 Bit, 16 Stellen genau)
  +4,446590812571219 *10^-323...+1,79769313486231 *10^308
- duda$ : STRING, Text-String (Zeichenkette, max. ca. 32767 Zeichen)
- preis@ : CURRENCY (nur in QuickBASIC ab 7.1/PDS), Währungstyp zum hoch-
  genauen Rechnen mit Geldbeträgen (64 Bit, 15 Vor-
  und 4 Nachkommastellen, intern als Ganzzahlen
  dargestellt) -9,22 * 10^14...+9,22*10^14
Gleitpunktzahlen sind Kommazahlen, bei denen das Komma an einer beliebigen
Stelle stehen kann. QBasic verwendet statt des Kommas einen Dezimalpunkt.
Negative Zahlen werden intern im Zweierkomplement dargestellt (Bei INTEGER-
Datentyp: -x = 2^16-x
z.B. -1 = FFFFhex = 1111 1111 1111 1111dual ).
- Zusätzliche Datentypen bei PowerBASIC (die Speichermöglichkeit für Strings
ist nur durch den freien Speicherplatz begrenzt {11/486}) :
- hugo? : BYTE, Byte (vorzeichenlose Ganzzahl, 8 Bit)

```

- hugo?? : WORD, 0...255  
Wort (vorzeichenlose Ganzzahl, 16 Bit)
- hugo???: DWORD, 0...65535  
Doppelwort (vorzeichenlose Ganzzahl, 32 Bit)
- hugo&& : QUAD, 0...4294967295  
vierfach lange Ganzzahl mit Vorzeichen (64 Bit)
- hugo## : EXT, +9,22 \*10^18  
Extended, Riesen-Gleitpunktzahl (80 Bit)
- hugo@ : FIX, +3,4\*10^-4932...+1,2\*10^4932  
BCD-Festpunktzahl (64 Bit; binär mit 4 Bit je Dezimalstelle kodiert)
- hugo@@ : BCD, +9,99\*10^-63...+9,99\*10^63  
BCD-Gleitpunktzahl (80 Bit; binär mit 4 Bit je Dezimalstelle kodiert)
- hugo@ : PTR, +9,99\*10^-63...+9,99\*10^63  
Pointer (Adresszeiger, 4 Bytes)
- hugo\$\$ : FLEX, Flexibler String (max 32750 Zeichen)
- Datendeklarationen (Definition der verwendeten Variablen mit ihrem Typ am Programmfang) sind nicht erforderlich, aber nützlich für den automatischen Typ-Check):
  - DIM [SHARED] <Variablenname ohne Typkennzeichen> AS <Typ> [, <Variable 2> AS...]
  - SHARED ==> auch Subroutinen können die Variable verwenden (Ohne SHARED sind sie dort unbekannt!). Umgekehrt kann das Hauptprogramm nur auf Variablen einer Subroutine zugreifen, die dort mit SHARED deklariert sind. Bei PowerBASIC kann DIM bei Verwendung von SHARED entfallen.
  - <Typ> = INTEGER|LONG|SINGLE|DOUBLE|STRING - (Datentyp, s. o.)
  - Beispiele: DIM anna AS LONG, otto AS SINGLE : anna& = 1.234  
DIM text AS STRING [\*12] - ["statischer" String mit der festen Länge von 12 Zeichen]
  - Bei einer mit DIM deklarierten Variablen kann das Typkennzeichen im nachfolgenden Programm weggelassen werden
  - Variablen, auf die das Hauptprogramm und ein Unterprogramm zugreifen können, müssen mit COMMON SHARED oder DIM SHARED als Globalvariable deklariert werden; siehe Abschnitt 'Geltungsbereich der Variablen' im Kapitel 'Allgemeines zu Subroutinen und Funktionen'.
- Standard-Datentypen für Variablen und FUNCTIONen festlegen:
  - {DEFINT|DEFLNG|DEFSNG|DEFDBL|DEFSTR} <Buchstabenliste> : Alle nicht über ein Typkennzeichen (% , & , ! , # , oder \$) definierten Variablen mit dem Anfangs-<Buchstaben> werden auf den Typ INTEGER|LONG|SINGLE|DOUBLE|STRING gesetzt. Bei der nachfolgenden Verwendung der Variablen kann das Typkennzeichen weggelassen werden. DEFxxx-Anweisungen müssen in SUBS und FUNCTIONs wiederholt werden: QBasic fügt dies automatisch ein.  
Beispiel 1: DEFINT A-Z 'alle Variablen ohne Typkennzeichen sind automatisch vom Typ INTEGER  
anna = 6 'Einfach lange Ganzzahl (INTEGER), % kann wegen DEFINT A-Z weggelassen werden  
anna! = 1.3 'Einfach lange Gleitpunktzahl (SINGLE)
  - Beispiel 2: DEFDBL D, I, X-Z 'Alle Variablen, deren Namen mit D = 10 ^ 100 'D, I, X, Y oder Z beginnt sind  
I = 10 ^ 100 'automatisch vom Typ DOUBLE  
Y = 10 ^ 100  
PRINT D, T, Y
- Typumwandlungen
  - implizit: "Implizit" heißt, dass die Typumwandlung automatisch bei einer Wertzuweisung von Variablen unterschiedlichen Typs stattfindet. Bei einer impliziten Gleitpunkt ==> Integer Wandlung wird auf die nächstgelegene Ganzzahl gerundet; Sonderfall: xxx.5 wird auf die nächste gerade Zahl gerundet. Beispiele:  
otto% = 5.5 ==> otto% := 6 (Sonderfall)  
otto% = 6.5 ==> otto% := 6 (Sonderfall)  
otto% = 2.678 ==> otto% := 3  
otto% = 23.42 ==> otto% := 23  
anna! = 3.51; otto% = anna! ==> otto% := 4  
otto% = 2; anna! = SQR(otto%) ==> anna! := 1.414214

- explizit: "Explizit" heißt, dass für die Typenumwandlung einer der folgenden speziell dafür vorgesehenen QBasic-Befehle verwendet wird:
  - FIX <Ausdruck> - erzeugt den ganzzahligen Anteil des numerischen Ausdrucks durch Abschneiden der Nachkommastellen. Es wird nicht gerundet im Gegensatz zur impliziten Typenumwandlung.  
Beispiele: FIX(12.45) ==> 12; FIX(-12.89) ==> -12
  - INT <Ausdruck> - gibt die größte Ganzzahl zurück, die kleiner oder gleich dem Ausdruck ist. Mit INT kann man auch Rundungen aller Art durchführen; siehe Kapitel 'Mathematische Funktionen...' unter INT und CINT.  
Beispiele: INT(12.45) ==> 12  
INT(-12.89) ==> -13
  - CDBL <Ausdruck> - numer. Ausdruck in DOUBLE-Gleitpunktzahl umwandeln (kann keinen numer. Stringausdruck (z. B. "2\*3") konvertieren!)
  - CSNG <Ausdruck> - numer. Ausdruck in SINGLE-Gleitpunktzahl umwandeln
  - CINT <Ausdruck> - Gleitpunktzahl auf eine INTEGER-Ganzzahl runden
  - CLNG <Ausdruck> - Gleitpunktzahl auf eine LONG-Ganzzahl runden
  - VAL (<String>) - String in Zahl umwandeln, z. B. VAL("2.34")
  - STR\$ (<Zahl>) - Zahl in String umwandeln, z. B. STR\$(2.34)
  - CQUD | CEXT | CFIX | CBCD - zusätzliche Typwandlungen für die speziellen PowerBASIC-Datentypen

//////////////////////////////////// Für Profis //////////////////////////////////////

- Der verfügbare Speicherplatz für Variablen und Stack lässt sich mit FRE und CLEAR abfragen und vergrößern; siehe Abschnitt 'Vorhandenen freien Speicherplatz für Variablen...' im Kapitel 'Direkter Speicherzugriff...' .

\*\*\*\*\*  
\* Fel der {3/12f} {6/102}  
\*\*\*\*\*

Ein Feld (engl. "Array") ist eine Aneinanderreihung von Variablen gleichen Typs unter einem gemeinsamen Variablennamen, den "Feldnamen". Die in einem Feld abgelegten Variablen nennt man "Feldelemente". Sie sind durchnummeriert: Jedem Feldelement ist eine Feldelementnummer, der "Index" zugeordnet. Der Zugriff auf ein Feldelement erfolgt über den Feldnamen und den in Klammern gesetzten Index. Mit DIM A%(100) wird beispielsweise ein Feld A%( ) mit 101 Feldelementen (Indices 0...100) angelegt (deklariert), das z. B. das Lebensalter von 101 Personen enthalten kann. Das Anzeigen des 3. Feldelement erfolgt dann beispielsweise über PRINT A%(2). Die einzelnen Elemente eines Feldes werden oft in FOR...NEXT-Schleifen bearbeitet; z. B. FOR i% = 0 TO 100: PRINT A%(i%): NEXT .

- Deklaration: DIM [SHARED] <Feldname> (<Anzahl Feldelemente%-1>) - z. B. DIM player\$(3) - Anlegen des Text-Feldes player\$( ) mit den 4 Feldelementen player\$(0)...(3)  
[SHARED] ==> Feld auch von SUBS u. FUNCTIONs ansprechbar  
oder : DIM <Name> (Nr. des ersten Elements) TO <Nr. des letzten Elements> - z. B. DIM player\$(1 TO 4). Hinweis: Bei PowerBASIC 'TO' durch ':' ersetzen!
- OPTION BASE 1 - Legt die erste Feldelement-Nummer (d. h. den kleinsten Index) aller Felder des Programms auf 1 statt auf 0 fest {6/207}.  
Beispiel: OPTION BASE 1: DIM anna%(5)  
==>Indices laufen von 1...5 statt von 0...4
- Wertzuweisung: <Feldname> (<Nr. des Feldelements%>) = <Wert>  
Beispiele: - DIM player\$(4) 'Feld mit 4 Elementen player\$(0...3)  
player\$(2) = "Tom" 'Wertzuweisung zum 2. Feldelement  
- DIM QuadratZahlen(1 TO 10) 'Feld mit 10 Elementen  
FOR i% = 1 TO 10  
QuadratZahlen(i%) = i% ^2: PRINT QuadratZahlen(i%)  
NEXT i%
- Feld zurücksetzen: ERASE <Feldname1> [, <Feldname2>... ] ; numerische Felder werden auf 0, Stringfelder auf den Leerstring "" gesetzt. Das Feld bleibt in voller Länge erhalten - außer bei dynamischen Feldern (s. u.).
- Max Feldlänge: Integer: 2^16-2 = 65534 Bytes, Long Integer: 2^16 Bytes
- LBOUND (<Feldname> [, Dimension%]) - liefert die kleinste Feldelement-Nr. (den kleinsten Index) des Feldes zurück (untere Grenze); bei mehrdimensionalen Felder die [Dimension] angeben. LBOUND wird z. B. von Subroutinen

benötigt, die beliebige Felder bearbeiten sollen (siehe SORT.BAS).

- **UBOUND** (<Feldname> [, Dimension%]) - liefert den größten Index des Feldes zurück (obere Grenze)
- Deklaration eines mehrdimensionalen Feldes mit einheitlichen Datentypen:
 

```
DIM anna% (1 TO 10, 1 TO 8) 'Deklaration (in PowerBASIC: (1:10, 1:8))
DIM anna% (10, 8) 'andere mögliche Form der Deklaration
anna% (3, 6) = otto% 'Wertzuweisung
```
- Deklaration eines mehrdimensionalen Verbund-Feldes gemischten Typs ("Anwenderdefinierter Typ"); die Typ-Deklaration muss im Hauptprogramm, darf nicht in SUBs oder FUNCTIONS erfolgen {11/269}:
 

```
TYPE quiz
frage AS STRING * 70 'Datentyp "quiz" deklarieren: Feld m je
antw1 AS STRING * 50 '3 String-Elementen (70, 50 und 50 Zei-
antw2 AS STRING * 50 'chen lang) und einem Integer-Element
oknr AS INTEGER 'Typ-Schlüsselwörter (STRING, INTEGER...):
'siehe Kapitel 'Datentypen...'
END TYPE

DIM geschichte (1 TO 20) AS quiz 'Anwenderdefiniertes Feld vom Typ
"quiz" deklarieren (auch in SUB oder FUNCTION möglich). Die Dimen-
sionierung kann auch dynamisch (d.h. erst während des Programmab-
laufs) erfolgen (z.B. '1 TO x%') {11/271}.
geschichte(1).frage = "Wer war der 1. Kanzler" 'Wertzuweisung;
geschichte(1).antw1 = "Erhard" 'Typkennzeichen
geschichte(1).antw2 = "Adenauer" 'weglassen
geschichte(1).oknr = 2

DIM puffer (2) AS quiz 'Wertzuweisung 'en bloc für ein gesamtes
puffer(1) = geschichte(1) 'Verbund-Feldelement ist auch möglich
'(großer Vorteil!!!)
```

Hinweis 1: Mit **LSET** <feld1(i)> = <feld2 (n)> kann man Inhalte zwischen 2 anwenderdefinierten Feldern kopieren

Hinweis 2: Anwenderdefinierte Felder lassen sich auch ineinander verschachteln ("Feld in Feld"). Beispiel:

```
TYPE quizerweitert
eintrag AS quiz
anzahl AS INTEGER
END TYPE
```

Hinweis 3: In anderen Programmiersprachen, z.B. C, werden die anwenderdefinierten Felder oft auch "Strukturen" oder "Records" genannt.

Hinweis 4: PowerBASIC kennt Verbundfelder erst ab V3.5! Diese lassen sich u.U. durch die so genannten Flex-Strings nachbilden; numerische Größen müssen hierbei durch STR\$|VAL in Strings gewandelt|rückgewandelt werden.

- Deklaration eines dynamischen Feldes (d.h. die Feldlänge lässt sich zur Programmlaufzeit verändern, und das Feld lässt sich wieder aus dem Speicher entfernen) {9/64+151} {6/215} {11/265}:
  - Variante 1: **DIM DYNAMIC** feld%(15) 'Feld konstanter Länge deklarieren  
ERASE feld% 'Feld aus dem Speicher entfernen
  - Variante 2: **INPUT** n%  
**DIM** feld% (n%):... 'Feld variabler Länge deklarieren; Feldlän-  
**INPUT** n% 'ge ändern und Feld initialisieren ==> alte  
**REDIM** feld%(n%) 'Daten werden gelöscht {11/265}, evtl. vorher in Hilfsfeld retten;  
'Anwendungsbeispiel: Siehe RANDOMNO.BAS  
**ERASE** feld% 'Feld aus dem Speicher entfernen
- **SSTATIC** | **SDYNAMIC** - über diese 'Metabefehle' lässt sich voreinstellen, ob alle nachfolgenden per DIM deklarierten Felder statisch oder dynamisch angelegt werden sollen (weniger gebräuchliche Befehle {11/268})
- Übergabe von Feldern an SUBs und FUNCTIONS: Felder können als Parameter an die SUB oder FUNCTION übergeben werden (mit leeren Klammern []). Sie sind dort jedoch erneut zu deklarieren {9/98} {6/223}; siehe Kapitel 'Allgemeines zu Subroutinen...'. Dort ist auch die Übergabe anwenderdefinierter Verbundfelder beschrieben.
- Hinweis zu PowerBASIC: Dort lassen sich über **ARRAY** {**SORT** | **SCAN** | **INSERT** | **DELETE**} Feldelemente komfortabel sortieren, suchen, einfügen und löschen.

```
*****
* Konstanten (CONST, DATA, READ)
*****
- CONST <Konstantenname> = Ausdruck [, <Konstantenname>= Ausdruck]... - Deklari-
tion von Konstanten. Die CONST-Deklaration muss vor Verwendung der
Konstanten erfolgen!
Beispiel: CONST pi# = 3.14159265358979 'Konstante vom Typ DOUBLE
KreisFlaeche# = r#^2 * pi# 'siehe {3/55}
Bei PowerBASIC sind nur INTEGER-Ganzzahlkonstanten möglich (CONST durch
'%' ersetzen, z.B. %anzahl = 37 statt CONST anzahl% = 37)
- DATA <Konstante1> [, <Konstante2>,...] - Deklaration von Konstanten, die mit
dem READ-Befehl eingelesen werden können {6/194}.
Beispiel: READ a%, text$ ==> a% = 2411; text$ = "otto"
...
DATA 2411, "otto"
DATA-Befehle können an beliebiger Stelle vor oder hinter dem READ-Befehl
stehen, nicht jedoch ein einer SUB oder FUNCTION. Es hat sich eingebür-
gert, die DATA-Zeilen ans Ende des Hauptprogramms zu schreiben. READ-
Befehle sind auch innerhalb von SUBs und FUNCTIONS verwendbar.
A C H T U N G! Hinter einem DATA-Befehl darf kein Kommentar stehen.
~~~~~ Sonst gibt es Fehlermeldungen beim Einlesen d. Konstanten!
- RESTORE - ermöglicht ein Wiederaufsetzen auf die erste per DATA deklarierte
Konstante (zum mehrmaligen Verwenden der DATA-Werte {9/25f}).
Zeilenweises RESTORE <MarkeS> ist auch möglich, wenn die DATA-Zeile
mit einer <MarkeS> versehen ist; siehe RESTORE.BAS und {5/57}
- Beispiele für Konstanten
(Hexadezimal-Zahl = Zahl zur Zahlenbasis 16, Oktal-Zahl = Zahl zur Basis 8
statt 10):
> -2.37 (SINGLE-Gleitpunktzahl) > "Egon" (Text-Zeichenkette, String)
> 235.988 E-7 (= 0.000235988; SINGLE) > &H5AB (Hexadezimal-Zahl 5AB)
> -2.5 D100 (DOUBLE-Gleitpunktzahl) > &o173 (Oktal-Zahl 173)
> -2.5# (DOUBLE-Gleitpunktzahl) > Dualzahl nicht vorgesehen !!
*****
* Mathematische Funktionen, Operatoren, Wertzuweisungen
*****
Wertzuweisungen
-----
- LET-Befehl: Der bei anderen Basic-Dialekten für Wertzuweisungen erforderliche
LET-Befehl ist möglich, aber nicht zwingend, z.B. ist x=1 identisch mit
LET x=1.
- SWAP <Variable 1>, <Variable 2> - Der Wert beider Variablen wird vertauscht
Der SWAP-Befehl eignet sich gut für Sortieralgorithmen aller Art (siehe auch
in der Online-Hilfe von QuickBASIC 4.5 zum SWAP-Befehl und Kapitel
'Tipps zu häufig vorkommenden Programmierproblemen').
- QBASIC initialisiert beim Programmstart alle numerischen Variablen mit dem
Startwert '0' und alle Strings mit dem Leerstring " " {9/31}
- ERASE <feldnameS> - setzt Felder auf den Startwert '0' bzw "" zurück (siehe
Kapitel 'Felder').
- CLEAR - initialisiert alle Variablen mit dem Startwert '0' bzw "" {11/251}
Mathematische Funktionen und Operatoren {3/115ff} (Priorität: siehe unten)
-----
- + - * / - Grundrechenarten. Division durch '0' führt zum Fehlerabbruch!
Bei einer Division von Integergrößen wird das Ergebnis auf die
nächste ganze Zahl auf- bzw. abgerundet.
- ^ - Exponentialzeichen, z.B. 2^10 ==> 1024
2^(1/3) ==> 3. Würzel aus 2 = 1.26
- \ - Ganzzahl-Division, schneidet den Rest ab,
z.B. 19\7 ==> 2; -19\7 ==> -2; 25.68\6.99 ==> 3
- x MOD y - Dividiert x durch y und gibt den Rest als Ganzzahl zurück
(ist x oder y eine Gleitkommazahl, so wird sie vorher gerundet)
z.B. 19 MOD 7 ==> 5; 10.4 MOD 4 ==> 2
19 MOD 6.7 ==> 5; -17.6 MOD 3.7 ==> -2
Der rückgelieferte Wert hat das gleiche Vorzeichen wie x
- SGN(x) - Vorzeichen von x, liefert -1|0|1, wenn x kleiner|gleich|größer
Null ist
- ABS (x) - Absoluter Betrag einer Zahl, z.B. ABS (-82) ==> 82
- INT (x!) - Integerwert erzeugen; liefert die nächstkleinere ganze Zahl,
z.B. INT (2.79) ==> 2; INT (-2.79) ==> -3
```

- **INT (x!+0.5)** - x! auf die nächstgelegene ganze Zahl auf-/abrunden, z. B. INT (5.67 + 0.5) ==> 6.  
Eine Rundung von Gleitpunktzahlen auf eine beliebige Anzahl von Vor- und Nachkommastellen ist folgendermaßen möglich:
  - INT (x \* 10^n + 0.5) / 10^n ==> x auf n Nachkommastellen auf-/ abrunden (bzw. n Vorkommastellen, wenn n negativ ist) {9/84} {3/127}
  - Beispiel 1: auf 3 Nachkommastellen runden: INT (1.23456 \* 10^3 + 0.5) / 10^3 ==> 1.235
  - Beispiel 2: auf ganze Hunderter runden: INT (320.5 \* 10^-2 + 0.5) / 10^-2 ==> 300
- **CINT (x)** - rundet eine Gleitpunktzahl zu einer einfach langen Ganzzahl (INTEGER). CINT rundet gemäß der Regeln der "Wissenschaftlichen Rundung". Dass heißt, ist die Nachkommastelle eine 5, so wird sie "abgeschnitten", wenn links davon eine gerade Ziffer steht, und sie wird aufgerundet, wenn links davon eine ungerade Ziffer steht.  
Beispiele:
  - CINT (12.49) -> 12
  - CINT (12.51) -> 13
  - CINT (12.50) -> 12
  - CINT (13.50) -> 14
  - CINT (13.5151 \* 100) / 100 -> 13.52 (Rundg. auf 2 Nachkommastellen)

Die Wissenschaftliche Rundung liefert genaue Ergebnisse bei statistischen Berechnungen, wie der Bildung von Mittelwerten usw. Bei der sogenannten "Kaufmännischen Rundung" wird im Gegensatz zur Wissenschaftlichen Rundung eine 5 als Nachkommastelle grundsätzlich immer aufgerundet. Hierzu verwendet man in QBasic den INT-Befehl (z. B. für Geldbeträge den Befehl INT (x + 0.5) ) oder - besser noch - den Befehl PRINT USING "#####.##"; x
- **CLNG (x)** - rundet eine Gleitpunktzahl zu einer doppelt langen Ganzzahl (LONG). Ansonsten hat der Befehl die gleiche Funktion wie CINT.
- **FIX (x!)** - Nachkommastellen abtrennen ohne Rundung  
z. B. INT (2.79) ==> 2 ; INT (-2.79) ==> -2
- **SIN|COS (x)** - Trigonometrische Funktionen Sinus und Cosinus {3/116} {11/322} (Winkel in Bogenmaß (Radian) eingeben: 2\*pi# im Bogenmaß entspr. 1 Vollkreis entspr. 360° im Gradmaß)  
z. B. cos 60° ausrechnen ==>  
pi# = 3.14159265358979  
COS(60 \* pi# / 180) ==> 0.5  
Ergebnis ist vom Typ DOUBLE.
- **TAN|ATN (x)** - Tangens und Arcustangens (x im Bogenmaß)
- **EXP (x)** - x-te Potenz zur Basis e (e^x); Ergebnis ist vom Typ SINGLE
- **LOG (x)** - Logarithmus zur Basis e (Natürlicher Logarithmus); Ergebnis ist vom Typ SINGLE {11/326}
- **SQR (x)** - Quadratwurzel (Ergebnis vom Typ SINGLE); negative Zahlen x führen zu Fehlerabbruch; Kubikwurzel: Siehe oben bei "^"
- **STR\$ (x)** - Zahl x in Zeichenkette umwandeln (ASCII-String) {3/120}  
z. B. STR\$(10.234) ==> " 10.234"
- **VAL (<String>)** - String in Zahl umwandeln, z. B. VAL ("2.43") ==> 2.43
- **HEX\$ (x)** - Zahl in Hexadezimalzahl-Zeichenkette umwandeln (Zahl zur Basis 16; z. B. HEX\$(100) ==> "64" = Hexadezimalwert der Dezimalzahl 100)
- Folgende mathematische Funktionen lassen sich durch Kombination der obigen Grundoperationen realisieren {9/87}:
 

|                                          |                               |
|------------------------------------------|-------------------------------|
| ArcSin (x) = ATN(x/SQR(-x*x + 1))        | Cot (x) = 1 / TAN (x)         |
| ArcCos (x) = -ATN(x/SQR(-x*x + 1))+pi#/2 | ArcCot (x) = - ATN(x) + pi#/2 |
| cosh (x) = ((e^x + e^(-x))/2             | sinh (x) = ((e^x - e^(-x))/2  |
| Loga (a, x) = LOG (x) / LOG (a)          | tanh (x) = sinh(x) / cosh(x)  |

Loga (a, x) = LOG (x) / LOG (a) 'Logarithmus von x zur Basis a  
nteWurzel (n, x) = x ^ (1/n) 'n-te Wurzel aus x
- **Mathematische Konstanten**
  - **Kreiskonstante pi**: CONST pi# = 3.14159265358979  
oder pi# = 4# \* ATN (1#) 'Näherung  
oder pi# = 355/113 'Näherung
  - **Eulersche Zahl e**: CONST e# = 2.718281829  
oder e! = EXP(1)
- **Abfangen von Zahlenüberläufen**: Ein Programm wird hart abgebrochen, wenn das Ergebnis einer mathematischen Funktion den Zahlenbereich des entsprechenden Datentyps überschreitet. Mit ON ERROR GOTO <Sprungmarke> kann man den harten

Abbruch vermeiden und eine Fehlermeldung mit Aufforderung zur Neueingabe anzeigen; siehe auch 'Fehlerbehandlung' im Kapitel 'Dateibearbeitung - Allgemeines...' und OVERFLOW.BAS .

**Vergleichsoperatoren** (Ergebnis = -1|0, wenn Bedingung erfüllt|nicht erfüllt)  
-----  
(Vergleich von Texten: siehe {9/57} )  
= | < | > - gleich | größer | kleiner  
- <> | <= | >= - ungleich | kleiner oder gleich | größer oder gleich

**Logikoperatoren** (auf INTEGER- und LONG-Integer-Zahlen anwendbar) {3/102}  
-----  
- **NOT x%** - Logische Invertierung aller Bits ("bitweises Komplement")  
- **x% AND y%** - Verundung aller Bits ("bitweise Konjunktion")  
- **x% OR y%** - Veroderung aller Bits ("bitweise Disjunktion")  
- **x% XOR y%** - Exklusiv-Oder-Verknüpfung aller Bits ("bitweise Antivalenz")  
- **x% EQU y%** - bitweise Äquivalenz (Ergebnis jew.=1, wenn beide Bits gleich) (Umkehrung der XOR-Funktion, wenig gebräuchlich)  
- **x% IMP y%** - bitweise Implikation (=NOT(x% AND (NOT y%)); wenig gebräuchl.)

**Priorität der Operatoren**  
-----  
höchste ==> ( ) | NOT  
AND | ^  
OR | \* / MOD  
= < > <> <= >= | niedrigste ==> + -  
-----

\*\*\*\*\*  
\* Textverarbeitung - Manipulation von Zeichenketten (Strings)  
\*\*\*\*\*

Siehe auch {3/108 f} {9/67} {6/249ff}  
Hinweis zur Textcodierung: In QBasic werden alle Textzeichen durch je ein Byte im DOS-ASCII-Code dargestellt. Die ASCII-Zeichencodes aller Zeichen finden Sie in der QBasic-Onlinehilfe unter "Hilfe -> Inhalt -> ASCII Zeichencodes". Sonderzeichen können Sie über die Tastatur mit [Alt + "0" + <Zeichencode>] eingeben.

- **t\$ = t1\$ + t2\$** - Strings zusammenfügen (z. B. "Es" + "el" ==> "Esel")
- **LEN (t\$)** - Länge des Strings t\$ (Anzahl Zeichen; Typ Integer)  
Beispiel: LEN("Esel") ==> 4
- **LEFT\$ (t\$, n)** - liefert n Zeichen links aus dem String t\$ zurück (n >=0)  
Beispiel: LEFT\$("Esel", 2) ==> "ES"
- **RIGHT\$ (t\$, n)** - liefert n Zeichen rechts aus dem String t\$ zurück (n >=0)  
Beispiel: RIGHT\$("Esel", 3) ==> "sel"
- **MID\$ (t\$, a [, n])** - Zeichen mitten aus einem String auslesen:  
Liefert n Zeichen ab dem a-ten Zeichen aus dem String t\$ zurück. Bei fehlendem n werden alle Zeichen ab dem a-ten Zeichen zurückgeliefert; a >= 0.  
Beispiel: MID\$("Schweineerei", 4, 5) ==> "weine"
- **MID\$ (t\$, a [, n]) = e\$** - Text ersetzen: Ersetzt n Zeichen im Text t\$ ab Zeichenposition a durch die ersten n Zeichen des Strings e\$. Bei fehlendem n werden alle Zeichen ab dem a-ten Zeichen ersetzt. Beispiel:  
t\$ = "Schafe": MID\$ (t\$, 4, 2) = "ul" ==> "Schule"
- **LCASE\$ (t\$)** - Groß- in Kleinbuchstaben umwandeln (nicht für Umlaute)
- **UCASE\$ (t\$)** - Klein- in Großbuchstaben umwandeln (nicht für Umlaute); z. B. Eingabe unabhängig von Groß-/Kleinschreibung gestalten:  
INPUT Text\$: IF UCASE\$(Text\$) = "ENDE" THEN END
- **LTRIM\$ (t\$)** - 'Führende Leerzeichen' vom Anfang eines Strings t\$ entfernen
- **RTRIM\$ (t\$)** - Leerzeichen vom Ende eines Strings t\$ entfernen
- **SPACES (15)** - String mit 15 Leerzeichen erzeugen
- **VAL (t\$)** - String in eine Zahl umwandeln  
Führende und nachfolgende Leerzeichen (Blanks) und Buchstaben werden überlesen und stören nicht.  
Beispiele: z\$ = "7.8": y! = VAL(z\$): PRINT y!\*2  
VAL (" 3.1E7cm") ==> 31000000  
{3/112} {9/67}
- **STR\$ (<Zahl>)** - Zahl in String umwandeln {3/120} {11/209}
- **HEX\$ (<Zahl>)** - Zahl in Hexa-Zahl-Zeichenkette umwandeln,

(Zahl zur Basis 16; z. B. HEX\$(100) = "64")

- **OCTS (<Zahl>)** - Zahl in Oktal-Zahl-Zeichenkette umwandeln (Zahl zur Basis 8; z. B. OCT\$(10) = "12")
- **INSTR ([<Beginn%>], <String1> <String2>)**
  - String suchen: sucht ab dem ersten [bzw. dem <Beginn%-ten] Zeichen in String1 nach String2 und gibt die Zeichenposition des ersten Auftretens zurück bzw. 0, wenn String2 nicht gefunden wird.
  - Beispiel: INSTR(5, "Mississippi", "si") ==> 7
- **STRINGS (<n>, <String>)**
  - gibt eine Zeichenfolge zurück, die <n>-mal hintereinander das erste Zeichen des <String> enthält, z. B.: PRINT STRINGS(80, "-") ==> Strich ziehen
- **ASC (<String>)** - gibt den ASCII-Code des ersten Stringzeichens zurück; z. B. ASC("ABCD") ==> 65 (ASCII-Code von "A")
- **CHRS (<Code%>)** - gibt das ASCII-Zeichen mit dem <Code%> zurück; siehe Kapitel 'Tastatureingaben'. Beispiel: CHRS(65) ==> "A"

\*\*\*\*\*  
 \* Textanzeige, Farben  
 \*\*\*\*\*

- **CLS** - löscht den Bildschirm (färbt ihn schwarz bzw. mit der über COLOR vorgegebenen Hintergrundfarbe ein; bei PowerBASIC-Versionen kleiner als V3.5 immer schwarz!).
- **LOCATE [<Zeile>] [, <Spalte>]** - setzt den Cursor auf die angegebene Bildschirmposition. Der Textbildschirm (Screen 0) hat 25 Zeilen mit je 80 Spalten bzw. die durch WIDTH definierte Anzahl von Zeilen und Spalten, siehe unten bei WIDTH. Bei Grafikbildschirmen hängt die maximale Anzahl der Spalten/Zeilen vom verwendeten Grafikmodus ab (siehe SCREEN 1...13 im Kapitel "Grafiken anzeigen").
- **LOCATE [<Zeile>], [<Spalte>], 1, 3, 5**
  - Cursor blinkend an gewählter Zeile u. Spalte setzen. Der Cursor erstreckt sich beim angegebenen Beispiel über die 3. bis 5. Pixelzeile.
  - Beispiele: LOCATE 1, 1 'setzt den Cursor in die linke obere Bildschirmcke
  - LOCATE 12, 39, 1, 1, 8 'Vollblock-Cursor in Zeile 12, Spalte 39
- **LOCATE , , 0** - Cursor wieder deaktivieren (unsichtbar machen; erscheint nach späteren LOCATE-Befehlen nicht mehr von selbst)
- **PRINT "text" [; ;]** - gibt Text an der Cursorposition aus und setzt den Cursor auf den Anfang der nächsten Zeile. Ausnahme: bei ";" am Ende bleibt der Cursor hinter dem Text stehen, bei "," wird der Cursor hinter die nächste freie Spalte im 14-ner Raster gesetzt, also auf eine der Spalten 15, 29, 43, 57 usw. (siehe {9/26}). Anführungszeichen " lassen sich über CHRS(34) einfügen; um "Hallo" mit Anführungszeichen anzuzeigen, schreibt man also PRINT CHRS(34)+"Hallo"+CHRS(34). Schreibfaule können in der Entwicklungsumgebung statt 'PRINT' auch '?' eingeben.
- **PRINT "Zahl"; anna%** - Gibt das Wort 'Zahl' und anschließend die in anna% gespeicherte Zahl aus (bei ',' statt ';' wird nach "Zahl" einem Tabulator eingefügt (Tabulator heißt Sprung hinter Spalte n\*14))
- **PRINT** ohne Zusatz - gibt eine Leerzeile aus
- **PRINT TAB(18); "Hallo"** - Cursor auf Spalte 18 setzen, dann "Hallo" ausgeben; die Zeichen zwischen der alten Cursorposition und Spalte 18 werden mit Leerzeichen überschrieben, d. h. gelöscht {9/27}.
- **PRINT SPC(10)** - 10 Spaces (Leerzeichen) ausgeben, z. B. zum Löschen von Bildschirmausgaben. Mit SPC lassen sich bei SCREEN 0 höchstens 79 Leerzeichen ausgeben (letztes Zeichen der Zeile nicht überschreibbar: SPC(80) funktioniert seltsamerweise nicht!). SPC ist nur in PRINT-Befehlen, nicht in Wertzuweisungen und Ausdrücken möglich (dort kann man nur SPACES verwenden).
- **PRINT STRINGS (<Anz>, <Text\$>)** - gibt Anz-mal das 1. Zeichen von Text\$ aus, z. B.: PRINT STRINGS (12, "\_") 'zeigt eine Linie an
- **PRINT USING <Maske\$>; <Ausdruck> [; <Ausdruck2> ; ...]** - formatierte Bildschirmausgabe mit einer Maske.  
 Die Maske gibt an, in welchem Format die Anzeige der Zahlen- oder Text-Ausdrücke erfolgen soll; siehe {3/68} {4/10f} {4/24} {9/30+43+141} {6/209} {11/84+308}.  
 PRINT USING ermöglicht die Anzeige von Tabellen. Nicht angezeigte Nachkommastellen werden "kaufmännisch" gerundet. Bezüglich des Aufbaus der <Maske\$>: Siehe QBasic-Onlinehilfe unter "Hilfe -> Index -> PRINT USING -> Format-Bezeichner".

Beispiel:  
 - PRINT USING "##.##"; 200/3 'Anzeige: 66.67 statt 66.6666 mit "Kaufmännischer Rundung" (4.2350 wird nach oben auf 4.24 gerundet)

Weitere Beispiele ("~" = Leerzeichen):  
 - maske\$ = "Der-Preis-betraegt~####.##-EUR" 'Maske mit Standardtext  
 a = 324.877  
 PRINT USING maske\$, a 'Anzeige: Der-Preis-betraegt~324.88-EUR  
 - maske\$ = "EURO-\*\*\*\*###.##" 'Anzeige: EURO-\*\*\*\*50.00  
 PRINT USING maske\$, 2542.23 'Anzeige: EURO-\*\*\*\*2542.23  
 'Führende Leerstellen werden mit "\*" fälschungssicher aufgefüllt  
 - maske\$ = "\~~~~~\~~~~###.##"  
 PRINT USING maske\$, "Tom"; 4.5 'Anzeige: Tom-----4.50  
 PRINT USING maske\$, "Sebastian"; 26.68 'Anzeige: Sebastia~26.68  
 'Textmaske in Backslashes "\": Zwischen den "\" wird eine Textmaske angegeben, hier 6 Leerzeichen entspricht einem Platzhalter für 8 Zeichen. Texte, die länger sind als die Textmaske werden abgeschnitten. Daher fehlt das "a" bei "Sebastian".

Beispiele für mehrere formatierte Anzeigen in einer Zeile:  
 - PRINT USING "###.##~###.##"; 100/3; 25.555 'Anzeige: 33.33~25.6  
 - PRINT USING "###.##"; 100/3; 25.555 'Anzeige: 33.33-25.56  
 - maske\$ = "Position###-Preis-###.##-EURO" 'Maske für 2 Variable  
 a=23; b=345.38  
 PRINT USING maske\$, a; b 'Anzeige: Position~23-Preis~345.38 EURO  
 - **VIEW PRINT <AnfZeile> TO <EndZeile>** - legt Ausgabefenster für Bildschirmausgabe fest; z. B.: VIEW PRINT 5 TO 24 'legt Ausgabefenster Zeile 5 bis 24 für die folgenden PRINT-Anweisungen fest. Die Ausgabe erfolgt dort rollierend; gut geeignet für die Anzeige von Tabellen, wenn die Tabellenüberschrift erhalten bleiben soll (siehe TASTCODE.BAS und JOYTEST.BAS).  
 - **COLOR [<Vordergr.farbe>] [, <Hintergr.farbe>]** - Bildschirmfarbe für Textbildschirm (SCREEN 0) angeben. Die Farben werden im 'DOS-Farbcode' angegeben (siehe unten stehende Tabelle). Vordergrundfarbe = Textfarbe  
 Beispiele: COLOR 0,7 = schwarze Schrift auf hellgrauem Grund  
 COLOR 14,1 = gelbe Schrift auf blauem Grund  
 COLOR 15,0 = Schwarz/Weiß-Bildschirm wiederherstellen  
 Der gesamte Bildschirm lässt sich durch ein anschließendes CLS mit der Hintergrundfarbe einfärben.

#### DOS-Farbcodes:

```

=====
|-----+-----+-----+-----+
| 0= schwarz   4= dunkelrot   8= grau       12= hellrot  |
| 1= blau      5= violett     9= hellblau   13= rosa    |
| 2= grün      6= braun/oliv  10= hellgrün  14= gelb    |
| 3= türkis    7= hellgrau   11= sehr helles 15= weiß    |
|-----+-----+-----+-----+
  
```

#### Anmerkungen zu den Farbcodes:

- 0...7 = dunkle Grundfarben, Addition von 8 ergibt jeweils die gleiche Farbe in hell
- In Screen 0 sind als Hintergrundfarben nur die ersten 8 Farben darstellbar und die Farbcodes 8...15 werden als Hintergrundfarben wie die Farbcodes 0...7 dargestellt!
- Eine Addition von +16 zum Farbcode der Vordergrundfarbe bewirkt blinkenden Text; funktioniert unter Windows nur im Vollbildmodus. COLOR 17,12 zeigt z. B. blaue Schrift blinkend auf hellrotem Hintergrund an.
- **POS (0)** - Systemvariable, liefert die aktuelle Spaltenposition des Cursors
- **CSRLIN** - Systemvariable, liefert die aktuelle Zeilenposition des Cursors
- **WIDTH <Spaltenzahl>, <Zeilenzahl>** - legt die Anzahl der Spalten und Zeilen fest. Beim Textbildschirm SCREEN 0 und VGA-Monitor z. B. Spalten x Zeilen= 40 x 25, 40 x 43, 40 x 50, 80 x 25, 80 x 43 oder 80 x 50; in SCREEN 12 auch 60 Zeilen möglich; bei EGA 80x25 oder 80x43, bei CGA 40x25 oder 80x25 Spalten x Zeilen möglich.  
 Bei WIDTH 40,25 wird im DOS-Fenster von Windows 3.1/95 nur ein halb breites Fenster dargestellt.

//////////////////////////////////// Für Profis //////////////////////////////////////

- WIDTH "SCRN:", <Spaltenzahl> - Breite der Ausgabezeilen festlegen {11/464}. Bei Spaltenzahl=40 erscheint unter Windows 3.1/95 ein halb breites Bildschirmfenster.
- SCREEN (<Zeile>, <Spalte> [, 1]) - Bildschirminhalt auslesen: Funktion, die den ASCII-Code des an der angegebenen Bildschirmposition angezeigten ASCII-Zeichens [bzw. dessen Farbwert] als INTEGER-Wert zurückliefert. Dieser muss vor einer erneuten Anzeige per PRINT mittels CHR\$ wieder in ein Textzeichen rückgewandelt werden (siehe {11/400} und SCREENRD.BAS).  
Beispiel: Erste Bildschirmzeile auslesen und in t\$ eintragen:  
FOR i% = 1 TO 80: t\$ = t\$ + CHR\$(SCREEN(1,i%)): NEXT i%
- WRITE <Variable1> [<Variable2>, ...] - Selten verwendete Methode, Datensätze auf dem Bildschirm anzuzeigen; Darstellung wie im Kapitel. 'Sequentielle Dateien' beschrieben (Strings in Anführungszeichen, Kommas zwischen den Variablen)

\*\*\*\*\*  
\* Tastatureingaben  
\*\*\*\*\*

- INPUT [";"] "Aufforderungstext" {;|,} <Variable> [<Variable2>...]
  - Kombinierte Bildschirm-Anzeige und Tastatureingabe. Nach Anzeige des Aufforderungstextes wartet der Befehl auf die Eingabe der Variablenwerte über die Tastatur, welche mit der Eingabetaste abgeschlossen wird. Gibt es mehrere Variablen, so muss der Anwender sie durch Kommas voneinander trennen. Die Eingabegrößen werden in den Variablen abgelegt. Das 1. Semikolon verhindert den Zeilenvorschub (Fortschaltung zur nächsten Zeile) nach dem Abschluss der Anwender-Eingabe. Wird das 2. Semikolon durch ein Komma ersetzt, so erscheint kein Fragezeichen hinter dem Aufforderungstext. Bei den Variablen kann es sich um numerische oder Textvariablen handeln. Kann eine Text-Variable Kommazeichen enthalten, so ist LINE INPUT statt INPUT zu verwenden (siehe unten im Abschnitt "Für Profis").

Beispiele: - INPUT "Wie lautet Dein Name?", name\$  
- INPUT "Wie alt bist Du"; alter% 'Hinter ..Du erscheint ein "?"  
- INPUT "Gib Deinen Namen und Dein Alter ein"; name\$, alter%  
'Das Anwender muss nach dem Namen ein Komma eintippen

- INPUT [";"] <Variable> - liest einen Wert von Tastatur ein und legt ihn in der Variablen ab. Es wird kein Aufforderungstext angezeigt. [";"] unterdrückt das Fragezeichen.

Beispiel: PRINT "Wie alt bist Du ? "; 'Aufforderungstext  
INPUT ";", alter% 'Das ";", unterdrückt ein zusätzl. Fragezeichen  
PRINT alter%

- SLEEP - wartet bis beliebige Taste betätigt wird (Unsaubere Methode, weil der 15 Zeichen umfassende Hardware-Tastaturpuffer nicht gelöscht wird! Bei dessen Überlauf können keine weiteren Tastenbetätigungen mehr erfasst werden, und es ertönt ein lästiges Piepsen über den PC-Speaker.)

- INKEYS - liest ein Zeichen von der Tastatur; im Gegensatz zu INPUT wird nicht automatisch auf eine Eingabe gewartet. Beispiele:

```
- IF INKEYS = CHR$(27) 'wenn Esc-Taste betätigt
- DO: LOOP UNTIL INKEYS = CHR$(27) 'warten bis Esc-Taste betätigt
- WHILE INKEYS = "" :WEND 'warten bis eine beliebige Taste betätigt;
' (" " = Leerstring keine Taste betätigt)
' Funktion kann auch durch x$ = INPUT$(1)
' (s. u.) oder Quick'n Dirty durch SLEEP
' ersetzt werden (s. o.)
```

- Beispiel für einfaches Tastamenü (Enter nicht erforderlich; siehe auch MENU1.BAS):

```
DO
CLS
PRINT <MenüText&>
DO: Taste$ = INKEYS: LOOP WHILE Taste$ = "" 'Warten bis Taste betätigt
SELECT CASE Taste$ 'Gehe zur Bearbeitung der aktuellen Taste
CASE "1": CALL Sub1 'Taste "1" betätigt
' --> Sprung zur Bearbeitungsroutine
CASE "2": CALL Sub2 'Taste "2" betätigt
CASE "3": CALL Sub3 'Taste "3" betätigt
CASE CHR$(27): END 'Programm beenden wenn Esc-Taste betätigt
END SELECT
LOOP 'neue Eingabe wenn andere Taste betätigt
```

- Tastencodes für die Sondertasten (siehe TASTCODE.BAS; die "Buchstaben" sind unbedingt als Großbuchstaben anzugeben):

| ++- Taste | --+- Code        | ---+ Taste    | ----- Code                       |
|-----------|------------------|---------------|----------------------------------|
| Enter     | = CHR\$(13)      | Cursor hoch   | = CHR\$(0) + "H"                 |
| Leertaste | = CHR\$(32)      | Cursor tief   | = CHR\$(0) + "P"                 |
| Backspace | = CHR\$(8)       | Cursor links  | = CHR\$(0) + "K"                 |
| Esc       | = CHR\$(27)      | Cursor rechts | = CHR\$(0) + "M"                 |
| Entf      | = CHR\$(0) + "R" | Bild hoch     | = CHR\$(0) + "I"                 |
| Entf      | = CHR\$(0) + "S" | Bild tief     | = CHR\$(0) + "Q"                 |
| Pos1      | = CHR\$(0) + "G" | F1... F10     | = CHR\$(0) + CHR\$(59)..... (68) |
| Ende      | = CHR\$(0) + "0" | F11.. F12     | = CHR\$(0) + CHR\$(133)... (134) |

Anwendungsbeispiele:  
IF INKEYS = CHR\$(0) + "H" THEN 'wenn Cursor hoch betätigt  
IF INKEYS = CHR\$(0) + CHR\$(60) THEN 'wenn F2-Taste betätigt  
Die ASCII-Codes der "normalen" Tasten finden Sie in der QBasic-Onlinehilfe unter "Hilfe -> Inhalt -> ASCII Zeichencodes"  
- WHILE INKEYS <> "" :WEND 'Tastaturpuffer leeren

//////////////////////////////////// Für Profis //////////////////////////////////////

- Ereignisgesteuerte Tastenbearbeitung (Tastenierrupt; siehe auch ONKEY.BAS):
  - ON KEY (<Tastennr. 1...31>) GOSUB <Name der Subroutines> - Ereignisgesteuertes Aufrufen einer Subroutine, wenn eine Taste betätigt wird; siehe ONKEY.BAS, {4/38}, {11/359}

- Die Subroutine muss als lokale Subroutine im Hauptprogramm definiert sein (siehe im Kapitel 'Lokale Subroutinen')
- Tastennr. = 1...10|30|31 ==> Funktionstasten F1... F10|F11|F12; F1 z. B. für Hilfefunktion verwendbar {11/362}
- Tastennr. = 11|12|13|14 ==> Cursorstasten Hoch|Links|Rechts|Tief
- Tastennr. = 15...25 ==> benutzerdefinierte Tasten (siehe Online-Hilfe zu ON KEY und {11/461f}). Diese Tasten werden wie folgt individuell belegt:

- KEY <Tastennr.>, CHR\$(<Tastenstatus%>) + CHR\$(<ScanCode%>)
  - Der Tastenstatus kennzeichnet Zusatzstastenbetätigungen:
    - 0 = keine Zusatztaste gedrückt | 8 = Alt
    - 1, 2, 3= Shift zusätzlich gedrückt | 32= NumLock aktiv
    - 4 = Strg zusätzlich gedrückt | 64= ShiftLock aktiv
  - Die Scancodes für die einzelnen Tasten findet man in der QBasic-Onlinehilfe unter <Hilfe | Inhalt | Kurzübersicht - Tastatur-Abfragecodes>

- Die Steuerung der Ereignisverfolgung erfolgt über KEY (<Tastennr.>) {ON | OFF | STOP} {11/463}
  - Tastennr.=0 ==> Die Steuerung der Ereignisverfolgung geschieht für alle Tasten gemeinsam
  - ON ==> Ereignisverfolgung aktivieren
  - OFF ==> Ereignisverfolgung deaktivieren
  - STOP ==> Ereignisverfolgung aktiviert, Ausführung erfolgt jedoch erst nach KEY ... ON

- Beispiel 1: KEY(1) ON 'Ereignisverfolgung für F1-Taste aktivieren  
ON KEY(1) GOSUB Hilfe 'Subroutine "Hilfe" aufruf. bei F1  
[Key(1) OFF] 'Ereignisverfolgung deaktivieren  
[Key(1) STOP] 'Überwachung der Taste unterbrechen, jedoch Tastenbetätigungen weiter erfassen und nach KEY(1) ON zur Wirkung kommen lassen.

- Beispiel 2: Key 15, CHR\$(0) + CHR\$(51) | (1) 'Komma- | Esc-Taste als benutzerdefinierte Taste 15 definieren  
ON KEY(15) GOSUB TuNix 'TuNix muss s.i. Hauptprogramm befinden  
KEY(15) ON  
...  
TuNix: PRINT "Ich Tu Nix": RETURN

- ON KEY(<Zahl 1...31>) GOTO <Sprungmarke\$> - Ereignisgesteuerte Tastenbearbeitung mit Direktsprung statt Aufruf einer Subroutine; ansonsten wie oben (wenig gebräuchlich {11/360}).

- Funktionstasten mit Zeichenketten belegen (für Menüs usw.; siehe MENU3.BAS) und {11/460}):

- KEY <Tastennr.>, <Zeichenkette\$> - Funktionstasten (Tastennr. 1...31: siehe oben bei ON KEY) mit einer Zeichenkette von max 15 Zeichen belegen
- KEY ON - Anzeige der (max. 6) Funktionstastenbelegungen in der unteren

- Bildschirmzeile aktivieren
- **KEY OFF** - Anzeige der (max. 6) Funktionstastenbelegungen in der unteren Bildschirmzeile deaktivieren
- **KEY LIST** - Komplette Liste aller Funktionstastenbelegungen anzeigen
- **<StringvariableS> = INPUTS(<n>)** - Spezialfunktion: Warten bis n Zeichen über die Tastatur eingegeben wurden (ohne Echo!); diese Zeichen werden in die Stringvariable eingetragen {9/72}. Hierfür gibt es nur wenige praktische Nutzanwendungen; höchstens vielleicht die folgende:
 

```
x=INPUTS(1) 'warten bis beliebige Taste betätigt
```

 Der Cursor kann über einen LOCATE-Befehl zur Anzeige gebracht werden.
- **LINE INPUT [;] ["Eingabeaufforderung";] <Stringvariable>**  
Einlesen einer kompletten Textzeile inklusive Kommas, welche sonst als Trennzeichen zwischen Eingabewerten dienen. Ein Fragezeichen wird nicht ausgegeben; [;] bewirkt, dass der Cursor in der Eingabezeile stehenbleibt {6/266}

\*\*\*\*\*  
 \* Grafiken anzeigen (geht nicht im Textmodus Screen 0)  
 \*\*\*\*\*

- Grafik-Bildschirmkoordinaten und ihre Verschiebung/ Skalierung:
  - Alle Koordinaten und Längenangaben werden normalerweise in Anzahl Pixeln angegeben (x, y = 0..max-1).
  - Beispiel: VGA-Bildschirmkoordinaten (x, y):
 

```
+-----+-----+
| (0, 0)   VGA   (639, 0) | ->x
| (0, 479) (639, 479) | v
+-----+-----+ y
```
  - Über **STEP** lassen sich bei vielen Grafikbefehlen relative Koordinaten aktivieren. Diese sind bezogen auf die momentane Position des Grafikcursors.
  - Eine Skalierung der Koordinaten ist mit dem WINDOW-Befehl möglich (multiplikative Beeinflussung des Maßstabs; siehe unten im Abschnitt 'Für Profis').
  - Zum Positionieren des Textcursors dient auch bei den Grafikbildschirmen (Grafikmodus >0) der - nicht pixelorientierte - LOCATE-Befehl, zum Festlegen der Spalten/Zeilenzahl der WIDTH-Befehl (siehe 'Textanzeige...').

- **SCREEN <Grafikmodus>** - Grafikbildschirm-Auflösung wählen {11/170} (SCREEN 0 vorbesetzt). Die gebräuchlichsten Grafikmodi sind:
  - SCREEN 0 = Textmodus, für alle Grafikkarten, läuft als einziger Bildschirmmodus auch problemlos im DOS-Teilfenster von Windows, 16 Farben, 8 Bildschirmseiten (0-7). Die anderen SCREEN-Modi laufen unter Windows nur im Vollbild.
  - SCREEN 1 = CGA/EGA/VGA-Karte, 320\*200 Grafik, 30\*25 Text, 4 aus 16 Farben, 1 Bildschirmseite (0) [2 Bits pro Pixel in 1 Ebene für GET/PUT],
  - SCREEN 2 = CGA/MCGA/EGA/VGA-Karte, 640\*200 Grafik, 80\*25 Text, 2 aus 16 Farben, 1 Bildschirmseite (0) [1 Bit pro Pixel in 1 Ebene für GET/PUT],
  - SCREEN 7 = EGA/VGA-Karte, 320\*200 Grafik, 40\*25 Text. Ruckelfreie Animationen auch auf langsamen Rechnern möglich, 16 Farben, 8 Bildschirmseiten (0-7) [1 Bit pro Pixel in 4 Ebenen für GET/PUT]
  - SCREEN 9 = EGA/VGA-Karte, 640\*350 Grafik, 80\*15 Text, bis 16 Farben, 2 Bildschirmseiten (0-1) [1 Bit pro Pixel (bei 16 Farben) in 4 Ebenen für GET/PUT]
  - SCREEN 11= VGA-Karte, 640\*480 Grafik, 80\*25|30|50|60 Text (Voreinstellung: 80\*30), 2 aus 256 Farben, gut geeignet für s/w-Grafiken 1 Bildschirmseite [1 Bit pro Pixel in 1 Ebenen für GET/PUT]
  - SCREEN 12= VGA-Karte, 640 x 480 Grafik, 80\*30|50|60 Text (Voreinstellung: 80\*30), 16 aus 256 Farben, 1 Bildschirmseite, [1 Bit pro Pixel in 4 Ebenen für GET/PUT],
  - SCREEN 13= VGA-oder MCGA-Karte, 320 x 200 Grafik, 40\*25 Text, 256 Farben, 1 Bildschirmseite, [8 Bits pro Pixel in 1 Ebene für GET/PUT],

- **SCREEN <Grafikmodus>, <Ausgabeseite>, <Anzeigeseite>** - Bildschirm-Ausgabeseite und Anzeigeseite umschalten. Die Anzahl der zur Verfügung stehenden Bildschirmseiten ist in der QBasic-Onlinehilfe unter <SCREEN | Bildschirmmodi> abfragbar; sie hängt vom Grafikmodus ab und kann bis zu 8 betragen. Die Verwendung mehrerer Seiten, das sogenannte "Page Flipping", unterstützt ruckelfreie Animationen {11/172}.

- **PCOPY <Quellseite>, <Zielseite>** - Inhalt einer Bildschirmseite in eine andere kopieren für Page Flipping {11/464}
- **COLOR** - Farbe verwenden; Syntax hängt vom verwendeten SCREEN ab.  
Beispiele: SCREEN 1 : **COLOR <Hintergrundfarbe>, <Farbpalette>** {11/185} SCREEN 7|8|9 : **COLOR <Zeichenfarbe>, <Hintergrundfarbe>** SCREEN 12|13 : **COLOR <Zeichenfarbe>**  
Bei Grafikbildschirmen (z. B. SCREEN 12) wird der Hintergrund durch **CLS: PAINT (x,y), <Farbcode>** eingefärbt (x,y beliebig).
- **LINE** - Linie oder Viereck zeichnen {11/175}; gestrichelte Linie durch Anhängen von [, &H<16-Bit-Hexa-Zahl>] möglich (endlos wiederholtes Pixelmuster: 0|1= Linienpixel nicht vorhanden|vorhanden {11/191})
  - **LINE [(x1,y1)-(x2,y2) [, <Farbcode>]** - farbige Linie von P1 nach P2
  - **LINE [(x1,y1)-(x2,y2) [, <Farbcode>], B [F]** - Viereck (Box) mit der Diagonalen P1-P2 zeichnen, [F=mit Farbe ausgefüllt]
  - **LINE [STEP (x1,y1) - STEP (x2,y2)...]** - dito mit relativen Koordinatangaben, d. h. bezogen auf die momentane Cursorposition
- Bei weggelassenem Anfangspunkt P1 (x1,y1) wird die momentane Position des Grafikcursors als Anfangspunkt verwendet.  
Beispiele (siehe auch KAESTEN.BAS und BOX.BAS):
  - SCREEN 12: **LINE (20, 30)-(600, 400), 4, , &HFOFF**  
'strichpunktierte Linie von (20|30) nach (600|400) in rot (4)
  - SCREEN 12: **LINE (20, 30)-(600, 400), 1, BF**  
'Rechteck mit den Eckpunkten (20|30) und (600|400) mit Farbe Blau gefüllt (Farbcode 1)
- **CIRCLE** - Kreis zeichnen {11/182}, auch für Ellipsen u. Kreisbögen
  - **CIRCLE [STEP] (x,y), <Radius> [, <Farbcode für Kreislinie>]** - Kreis zeichnen. [STEP] definiert die Kreismittelpunkt-Koordinaten x und y als relative Koordinaten, bezogen auf die momentane Cursorposition. Der Kreis lässt sich mit Farbe füllen über **PAINT (x,y), <Farbcode>**  
Beispiel: SCREEN 12: **CIRCLE (320, 240), 150, 2**  
'Kreis mit Mittelpunkt (320|240) und Radius 150  
'zeichnen, Randfarbe grün (Farbcode 2)
  - **CIRCLE (x,y), <Radius>, [Farbcode],,, <Faktor>** - Ellipse mit Stauchungs-Faktor Höhe/Breite zeichnen {11/182+198}; die Ellipse passt immer in den Kreis mit dem angegebenen Radius hinein (Faktor < 1 ==> Breite = Radius; Faktor > 1 ==> Höhe = Radius)  
Beispiel: SCREEN 12: **CIRCLE (320, 240), 150, 2, , , .4**  
'Ellipse mit Mittelpunkt (320|240), Breite=150  
'mit grüner Randfarbe (2) und Höhe= 60  
'(150 \* 0.4)
  - **CIRCLE (x,y), <Radius>, [Farbcode], <Anfangswinkel>, <Endwinkel> [, <Faktor>]** - Kreisbogen [Ellipsenbogen] zwischen Anfangs- und Endwinkel zeichnen (Winkelangaben im Bogenmaß (d. h. in Radian: 3,14 Radian=pi=180°), oben = 0°, wird im Uhrzeigersinn gezeichnet {11/183}).  
Beispiel: SCREEN 12  
**CIRCLE (400, 200), 180, 4, 3.14, 3.14 / 2, .4**  
'3/4-Ellipsenbogen um den Punkt (400,200) mit der  
'Breite 180 Pixel von 180° (Pi) nach 90° (Pi/2)  
'ziehen; Randfarbe rot (Farbcode 4),  
'Höhe = 180\*0.4 = 72 Pixel
- Bei negativen Winkelangaben entsteht eine Tortengrafik, und die beiden Enden des Kreis-/Ellipsenbogens werden mit dem Mittelpunkt verbunden.  
Beispiel: SCREEN 12: **CIRCLE (100, 100), 75, 12, -3.14, -3.14/2**  
'Zeichnen eines 3/4 Tortenstücks mit roter Randlinie  
'(Farbcode 12) von -180° nach -90° (=270°)
- **PAINT [STEP] (x,y), <Randfarbe>** - vorher mit LINE und/oder CIRCLE begrenzte Fläche mit der Randfarbe einfärben (STEP macht die Koordinaten relativ, d. h. bezogen auf die momentane Cursorposition {11/189})  
Beispiel: SCREEN 12  
**LINE (20, 10)-(300, 200), 5** 'Violettes (5) Dreieck  
**LINE -(600, 50), 5** ' (20|10)-(300|200)-(600|50)  
**LINE -(20, 10), 5** 'zeichnen und mit der Rand-  
**PAINT (280, 100), 5** 'farbe Violett ausfüllen
- **PAINT [STEP] (x,y), <Füllfarbe>, <Randfarbe>** - Fläche einfärben bis die Randlinie mit der angegebenen Randfarbe erreicht wird
- **PAINT [STEP] (x,y), <MusterS>, <Randfarbe>** - Fläche mit Muster\$ ausfüllen bis die Randlinie mit der angegebenen Farbe erreicht wird. Muster\$ wird im Binärcode (d. h. Bit für Bit) interpretiert und 1-Positi-

- onen mit der aktuellen Zeichenfarbe in horizontal wiederholten Reihen eingefärbt (siehe {11/190} und MUSTER.BAS).
  - **PSET [STEP] (x,y) [, <Farbcode>]** - einen Bildpunkt (Bildschirmpixel) malen. STEP macht die x/y-Koordinaten relativ.
  - **PRESET [STEP] (x,y)** - Bildpunkt löschen (mit Hintergrundfarbe übermalen)
  - **DRAW <Befehls-StringS>** - Polygonzug aus aneinandergesetzten Linien zeichnen. Verketteter Befehl zum Zeichnen aneinanderhängender Linien mit einem gedachten Zeichenstift, der gleichzeitig dem Grafikkursor entspricht (Syntax ähnlich dem Sound-PLAY-Befehl).  
Teilelemente des Befehls-StringS: {11/218}
  - <RichtungsbuchstabeS> [<n%>] - bewegt den Zeichenstift um 1 [oder n%] Pixel in die durch den Richtungsbuchstaben definierte Richtung und zeichnet eine entsprechende Linie.
- ```

      H U E
      \ | /
Richtungsbuchstaben: L-+-+R
      / | \
      G D F

```
- **M [+|-] x, [+|-] y** - bewegt den Zeichenstift auf [um] die Koordinaten x, y und zeichnet eine entsprechende Linie. + bzw. - bewirkt eine Bewegung um relative Koordinaten, d.h. bezogen auf die aktuelle Cursorposition.
  - **B** - Präfix: Zeichenstift heben und ohne zu zeichnen bewegen.
  - **N** - Präfix: Nach dem Zeichnen Zeichenstift wieder auf Ausgangsposition setzen
  - **C <n%>** - Zeichenfarbe setzen
  - **A <n%>** - Zeichenstift um n%\*90° entgegen dem Uhrzeigersinn drehen (n%=1, 2 oder 3) bzw. das 'Zeichenblatt' unter dem Zeichenstift um 90° im Uhrzeigersinn drehen. Die Richtungsbuchstaben ändern ihre Wirkungsrichtung entsprechend.
  - **TA <n%>** - Zeichenstift um n° drehen (n%=-360..+360). Die Richtungsbuchstaben ändern ihre Richtung entsprechend. {11/221}
  - **An%|Bn%** - Objekt um n% Grad rotieren | Zeichenfarbe setzen
  - **P n1%, n2%** - Füll- und Randfarbe eines Objektes setzen
  - **S n%** - Längeneinheit für Zeichenstiftbewegung setzen (4 entspricht 1 Pixel)
  - **Beispiel 1:** Dreieck zeichnen durch Verbindung der 3 Punkte (200, 50), (250, 50) und (250, 20) mit roten Linien {11/221} :  
SCREEN 12: DRAW "C4 BM200,50 R50 U30 M200,50"
  - **Beispiel 2:** Haus aus blauen Linien zeichnen:  
SCREEN 9  
Haus\$ = "C1 TAO BMI80,150 R50 U50 L50 D50 R50 BMI80,100"  
Haus\$ = Haus\$ + " E25 F25"  
DRAW Haus\$

//////////////////////////////////// Für Profis //////////////////////////////////////

- **PALETTE <Farbkennziffer>, <Farbcode>** - Einer bestimmten Kennziffer einen neuen Farbcodes zuordnen. Die Zahlenbereiche der Farbkennziffer und des Farbcodes hängen vom verwendeten SCREEN-Modus ab. Die Farbkennziffer kann im Bereich 0... <Anzahl darstellbare Farben - 1> liegen. Der Farbcode ist z.B. beim SCREEN 13 ein RGB-Code (Rot/Grün/Blau-Farbmischung); siehe RGBFARBE.BAS und {11/186}.
- **PALETTE USING <Feld[ (<Startindex>)]>** - Farbpalette einem Feld zuweisen, Feld muss vorher mit Farbcodes & gefüllt werden {11/186}
- **POINT (x, y)** - Funktion, die den Farbcode des Bildschirmpunktes (x, y) zurückliefert
- **VIEW [SCREEN] (x1,y1) - (x2,y2)** - Bildschirmausschnitt als aktuelle Grafikfläche definieren, die mit CLS selektiv gelöscht werden kann.
  - Dieser Befehl ist gut geeignet zum Zeichnen von kleinen animierten Grafiksymbolen (Sprites), die mit GET/PUT angezeigt und abgespeichert werden sollen; siehe {11/197..201} und GETPUT1.BAS.
  - Bei VIEW ohne SCREEN beziehen sich alle in den nachfolgenden Befehlen verwendeten Koordinaten auf die linke obere Ecke des Bildschirmausschnitts (d.h. der Punkt (0,0) ist identisch mit (x1, y1).
  - Bei Angabe von SCREEN beziehen sich alle Koordinaten nach wie vor auf die linke obere Ecke des Gesamtbildschirms.
- **VIEW** - (ohne Parameter): Die obige Bildschirmausschnitt-Definition wieder aufheben {11/198}

- Mit **GET/PUT** lässt sich Grafik in einem RAM-Bildfeld speichern/ Bildfelddaten als Grafik anzeigen. Damit kann man Bildelemente bequem in RAM-Feldern speichern und schnell auf den Bildschirm ausgeben - ohne langwieriges Neuzeichnen (siehe {11/202+205} und GETPUT1.BAS):

- **DIM <Bildfeldname> (<laenge>)** - INTEGER-Bildfeld zur Ablage eines Bildelements deklarieren. Die erforderliche <laenge> des Feldes hängt vom verwendeten SCREEN-Grafikmodus und der Hoehe/ Breite des Bildschirmausschnitts wie folgt ab (siehe {11/205} und QBasic-Online-Hilfe unter <Hilfe | Index | PUT-Anweisung (Grafik) | Bilddatenfelder und Kompatibilitäten>):

```

+-----+
| laenge& des für eine Grafik-Get/PUT-Operation benötigten Feldes = |
| (4 + Hoehe*Ebenen*INT((Breite*BitsProPixel/Ebenen + 7)/8))\2 + 1 |
+-----+

```

- mit - Hoehe = Höhe des Bildschirmausschnitts y2-y1+1 (siehe GET)
- Breite = Breite des Bildschirmausschnitts x2-x1+1 (siehe GET)
- Ebenen = Anzahl der Farbebenen (abhängig vom Grafikmodus, siehe oben bei den SCREENS 1...13)
- BitsProPixel = Speicherbedarf je Bildschirmpixel (abhängig vom Grafikmodus (siehe oben bei den SCREENS 1...13))

- **GET [STEP] (x1,y1) - [STEP] (x2,y2), <Bildfeldname>** - Rechteckigen Ausschnitt des Anzeigebildschirms in Bildfeld einlesen. (x1,y1)= obere linke, (x2,y2)= rechte untere Ecke. STEP macht die Koordinaten relativ. Die erforderliche Länge des Bildfeldes ergibt sich aus der Formel im obenstehenden Kasten.

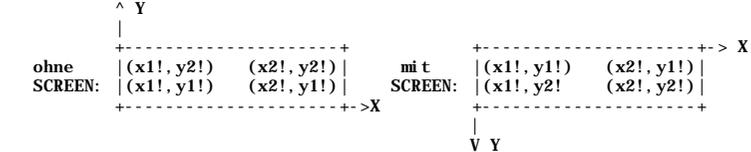
- **PUT [STEP] (x1,y1), <Bildfeldname>** - Grafikinformation aus dem Bildfeld auf den Bildschirm an der durch (x1,y1) gekennzeichneten Stelle zur Anzeige bringen. (x1,y1)= Koordinaten der linken oberen Ecke. Die alte auf dem Bildschirm angezeigte Grafikinformation wird vorher gelöscht. STEP macht die Koordinaten relativ. Die erforderliche Länge des Bildfeldes ergibt sich aus der Formel im obenstehenden Kasten.

- **PUT (x1, y1), <Bildfeldname>, {PSET|PRESET|AND|OR|XOR}** - Entspricht dem obigen PUT-Befehl, jedoch wird die alte Anzeigeeinformation nicht gelöscht, sondern wie folgt mit der Grafikinformation des Bildfeldes verknüpft (siehe {11/207} und GETPUT2.BAS):

- PSET** = löscht vorhandenen Bildausschnitt, fügt neues Bild ein
- PRESET** = löscht vorhandenen Bildausschnitt, invertiert neues Bild
- AND** = mischt neues mit vorhandenem Bild: Nur Bildpunkte, die im alten und neuen Bild gezeichnet sind, erscheinen auf dem Bildschirm
- OR** = überlagert vorhandenes mit neuem Bild: Alle gezeichneten Bildelemente des alten und des neuen Bildes werden dargestellt (z.B. für transparente Sprites!!)
- XOR** = überlagert vorhandenes mit neuem Bild: Wie OR, jedoch bleiben diejenigen Bildpunkte dunkel, die im alten und im neuen Bild Zuelemente enthalten.

- **WINDOW [SCREEN] (x1!,y1!) - (x2!,y2!)** - Skalierung der x/y-Koordinaten: Die nachfolgenden Grafikbefehle verwenden nicht Pixelkoordinaten, sondern 'virtuelle' Koordinaten. Die durch virtuelle Koordinaten angegebenen Punkte (x1!,y1!) und (x2!,y2!) entsprechen den Eckpunkten des Gesamtbildschirms bzw. des mit einem vorangegangenen VIEW-Befehl definierten Bildschirmausschnitts (siehe oben). Siehe auch {11/213} und SINUS.BAS.

- Bei weggelassenem SCREEN kehrt sich die Wirkungsrichtung der x/y-Koordinaten um (y\_unten= 0, y\_oben= max; wie in der Mathematik üblich).
- Bei Verwendung von SCREEN hat die y-Achse die gleiche Wirkungsrichtung wie bei normalen Pixelkoordinaten. Beispiel für VGA-Bildschirm:



Zur Umrechnung von Pixel- in virtuelle Koordinaten und umgekehrt stehen die Befehle POINT und PMAP zur Verfügung:

- **POINT <Modus%** - liefert als Zielwert die aktuelle Position des Grafikkursors in Pixelkoordinaten bzw. virtuellen Koordinaten (siehe WINDOW-Befehl) gemäß der folgenden Tabelle zurück {11/216}:

| Modus% | Startwert                | Zielwert                 |
|--------|--------------------------|--------------------------|
| 0      | x-Koordinate virtuell *) | x-Koordinate, Pixel      |
| 1      | y-Koordinate virtuell *) | y-Koordinate, Pixel      |
| 2      | x-Koordinate, Pixel      | x-Koordinate virtuell *) |
| 3      | y-Koordinate, Pixel      | y-Koordinate virtuell *) |

\*) = virtuelle Koordinaten = mittels WINDOW-Befehl skalierte (d.h. gestauchte oder gedehnte) Koordinaten.

Der Startwert ist für den POINT-Befehl ohne Bedeutung.

- **PMAP (<Startwert>, <Modus%**) - Rechnet den Startwert entsprechend obiger Tabelle um und liefert den Zielwert als Ergebnis zurück {11/216}.
- **BSAVE** und **BLOAD** - Bildschirm Inhalte in Datei ablegen bzw. aus Datei lesen und anzeigen (siehe Beschreibung von BSAVE/BLOAD im Kapitel 'Direkter Speicherzugriff...')

\*\*\*\*\*  
\* Sound aus PC-Speaker ausgeben  
\*\*\*\*\*

- **BEEP** - einen Piepston erzeugen (identisch mit PRINT CHR\$(7) 'Ausgabe des Bell-Zeichens')

- **SOUND <FrequenzInHerz%>, <DauerInSystemtakten%>** - einfache Art der Soundausgabe: Es wird ein Ton mit der angegebenen Frequenz der Dauer in Anzahl von Systemtaktungen à 56 ms (=0,056 s) ausgegeben. Der Sound-befehl ist zur Erzeugung von Soundeffekten aller Art gut geeignet, weil sich die Frequenz und die Dauer der Töne in FOR-Schleifen verändern lassen; siehe Sirene in {6/134}, {11/224}, KLAVIER.BAS und MUSIK.BAS).

Beispiele: - SOUND 2000, 6 '2000Hz-Ton 6\*55ms=330ms lang spielen  
- DO: SOUND 192, 0.5: SOUND 188, 0.5 : LOOP 'Motorengeräusch

- **PLAY <Befehls-String\$>** - Komfortable Ausgabe von Musikstücken über den PC-Speaker (siehe {11/222} und KLAVIER.BAS).

Teilelemente des Befehls-String\$:

----- Noten spielen, Oktave festlegen -----

- **M[F|B]** - alle folgenden Noten in Vordergrund|Hintergrund abspielen (Foreground|Background). 'Vordergrund' bedeutet, dass mit der Abarbeitung der Folgebefehle solange gewartet wird, bis der PLAY-Befehl komplett abgearbeitet worden ist. 'Hintergrund' bedeutet, dass während des Spielens das Programm fortgesetzt wird. Vorbesetzung= MF

- **{A|B}...{G|}** - Note a, h, c, d, e, f oder g der Tonleiter in der aktuellen Oktave spielen

- **O<n%** - aktuelle Oktave für die folgenden Noten festlegen (n%=0..6)

- **N<n%** - einen Ton aus dem gesamten 7-Oktav-Bereich spielen (n%=0..84, 0=Pause)

- **<** - eine Oktave erhöhen, gilt für alle nachfolgenden Töne

- **>** - eine Oktave erniedrigen, gilt für alle nachfolgenden Töne

----- Tonlänge, Tempo, Pausen -----

- **L<q%** - Länge der nachfolgenden Töne festlegen (q=1-64; Tonlänge = 1/q; 1 ist eine ganze Note; Vorbesetzung: q = 4 ==> 1/4 Note)

- **P<q%** - Pausendauer zwischen den nachfolgenden Tönen festlegen (q=1-64; Pausendauer = 1/q; Vorbesetzung: q = 4 ==> 1/4 Note)

- **T<q%** - Tempo der nachfolgenden Noten in Viertelnoten/min festlegen; (q=32-255); Vorbesetzung: q= 128

----- Folgezeichen (Suffixe) für Einzelnoten -----

- **{+|#}** - Suffix: Die vorangehende Note um einen Halbtonschritt erhöhen

- **-** - Suffix: Die vorangehende Note um 1 Halbtonschritt erniedrigen

- **.** - Suffix: Die vorangehende Note 1,5 mal so lang spielen

----- Staccato und Legato -----

- **MS** - alle nachfolgenden Noten in Staccato spielen (kurz und abgehackt, nicht mit dem nächsten Ton verbunden)

- **ML** - alle nachfolgenden Noten in Legato spielen (lang und getragen, mit der nächsten Note verbunden)

- **MN** - alle nachfolgenden Noten wieder normal spielen (nicht Staccato oder Legato)

```
----- Beispiel -----
- PLAY "MB ML T160 01 L2 gdec P2 fedc" 'Big-Ben-Schlag
im Hinter- | | | | | | | | | | | | | | | | | | | | | |
grund --+ | | | | | | | | | | | | | | | | | | | | |
Legato -----+ | | | | | | | | | | | | | | | | | | |
Tempo 160 -----+ | | | | | | | | | | | | | | | | | |
1. Oktave -----+ | | | | | | | | | | | | | | | | | |
                                         letzte 4 Noten
                                         1/2 Notenlänge Pause
                                         erste 4 Noten
                                         Notenlänge: 1/2 Note
```

//////////////////////////////////// Für Profis //////////////////////////////////////

- **ON PLAY (<Notenanzahl%> GOSUB <Marke\$>** - Ereignisgesteuertes Anspringen der lokalen Subroutine <Marke\$>, wenn der PLAY-Notenpuffer weniger noch ungespielte Noten als die angegebene <Notenanzahl%> enthält (z. B. =2). Die Subroutine enthält normalerweise einen Play-Befehl mit 'Notennachschub' für lange Hintergrundmusiken {11/370}

- **PLAY {ON|OFF|STOP}** - Ereignisverfolgung für Notenpufferauswertung aktivieren | deaktivieren | unterbrechen mit Speicherung

- **PLAY(0)** - liefert die Anzahl der gerade im PLAY-Notenpuffer stehenden noch ungespielten Noten zurück {11/370}

\*\*\*\*\*  
\* Joystickabfrage  
\*\*\*\*\*

Siehe auch JOYTEST.BAS.

- **STICK(0) | (1) | (3)** - X|Y-Achse| Schubregler abfragen, Rückgabewert 255 ... 0

- **STRIG(1) | (5)** - Rückgabewert -1 = Feuerknopf A|B betätigt (bei PowerBASIC vorher mit STRIG ON die Ereignisverfolgung aktivieren)

- **ON STRIG (<Knopfnr%> GOSUB <Sprungmarke\$>** - Ereignisgesteuertes Anspringen der lokalen Subroutine <Sprungmarke\$> bei Drücken eines Joystick-Knopfes.

Knopfnr% 0 | 4 = unterer | oberer Druckknopf. Siehe auch JOYINTR.BAS und {11/373}.

- **STRIG {ON|OFF|STOP}** - Ereignisverfolgung für Joystick-Knöpfe aktivieren|deaktivieren|unterbrechen mit Speicherung

Anmerkung 1: QBasic unterstützt keine Joysticks, die am USB-Port hängen, sondern nur Joysticks, die an den Standard PC-Gameport angeschlossen sind.

Anmerkung 2: Achtung: die Joystick-Abfragewerte werden bei gleichzeitiger Sound-Ausgabe auf den PC-Speaker verfälscht!

Anmerkung 3: Abfrage des 2. Joysticks B: Siehe QBasic-Onlinehilfe unter STRIG und STICK sowie {11/373}

\*\*\*\*\*  
\* Wartezeiten erzeugen und Datum/ Uhrzeit bearbeiten  
\*\*\*\*\*

- **SLEEP [<n%>]** - Wartezeit n sec einlegen (nur ganze Sekunden); Die Wartezeit wird bei Betätigung einer beliebigen Taste vorzeitig abgebrochen. Beispiel: SLEEP 2 '2 sec warten; SLEEP ist bei PowerBASIC erst ab V3.5 vorhanden (bei älteren Versionen durch DELAY ersetzen). Bei SLEEP mit Parameter '0' oder ohne Parameter wird bis zur nächsten Tastenbetätigung gewartet, der Tastaturpuffer jedoch nicht geleert.

- **TIMER** - Systemuhr, liefert die seit Mitternacht vergangenen Sekunden zurück. Der TIMER wird 18,2 mal je Sekunde um den Wert 0,056 sec erhöht. Kürzere Wartezeiten als 0,056 sec lassen sich mit der TIMER-Funktion also nicht realisieren. Der TIMER liefert Gleitpunktwerte vom Typ SINGLE zwischen 0.000 bis 86399.944 (entspricht den während der 24 Stunden von 00:00:00 h bis 23:59:59.944 h abgelaufenen Sekunden). Bei der Realisierung von Wartezeiten, Stoppuhren und Countdown-Timern ist der Rücksprung des TIMERS vom Maximalwert 86399.944 auf 0.000 um Mitternacht zu berücksichtigen.

Beispiel zur Erzeugung einer feinaufgelösten Wartezeit von 0,5s: starttime! = TIMER 'seit Mitternacht abgelaufene Zeit in s  
DO: LOOP UNTIL TIMER > starttime! + .5

Zur Bildung von kürzeren Wartezeiten unter 0,056 sec kann man den programmierbaren Intervall-Timer (PIT) 8253/8254 verwenden (siehe Abschnitt 'Zugriff auf I/O-Ports' im Kapitel 'Direkter Speicherzugriff...').

- **MITIMER** - Nur bei PowerBASIC vorhanden: Mikrotimer mit einer Auflösung von 1µs

- **DATES** - Datum ausgeben als String im Format MM-TT-JJJJ, z. B. "04-29-1999" (Umwandlung in deutsches Format: Siehe DAT-ZEIT.BAS). Systemdatum änderbar durch **DATES = <Datum-String\$>**

- **TIMES** - Uhrzeit ausgeben als String im Format HH:MM:SS, z. B. "18:58:12".  
Systemzeit änderbar durch **TIMES = <Uhrzeit-String>**

//////////////////////////////////// Für Profis //////////////////////////////////////  
Der System-Timer lässt sich auch ereignisgesteuert bearbeiten (siehe ONTIMER.BAS). Diese Funktion wird aber nur selten verwendet:

- **ON TIMER (<AnzahlSekunden>) GOSUB <Marke\$>** - Ereignisgesteuert (abhängig vom Timerinhalt) wird alle <AnzahlSekunden%> die lokale Subroutine <Marke\$> angesprungen; AnzahlSekunden% kann einen ganzzahligen Wert zwischen 0 und 86399 annehmen (entspricht den 24 Stunden von 00:00:00h ... 23:59:59h) {11/366}.
- **TIMER {ON|OFF|STOP}** - Ereignisverfolgung für Timer aktivieren|deaktivieren |unterbrechen mit Speicherung.

\*\*\*\*\*  
\* Zufallszahlen erzeugen {9/85}  
\*\*\*\*\*

- **RANDOMIZE TIMER** - Zufallsgenerator auf Systemuhr-abhängigen, d.h. bei jedem Programmstart anderen Startwert setzen. Der Zufallsgenerator erzeugt bei gleichem Startwert immer dieselbe Reihe von Zufallszahlen

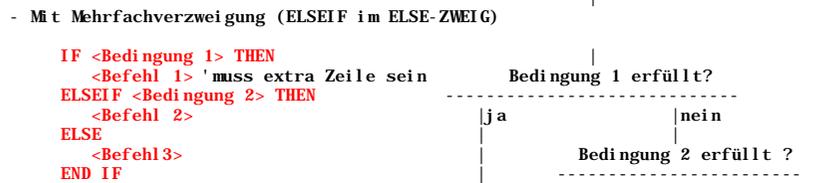
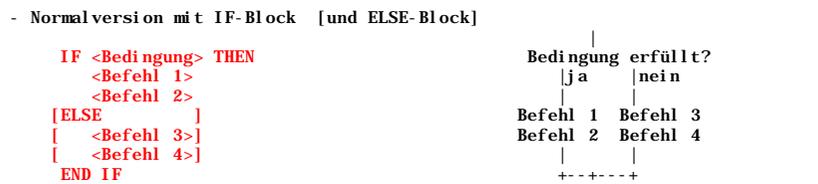
- **RND** - liefert eine Zufallszahl vom Typ SINGLE zwischen 0 und 0.9999999;
- Beispiele: **RANDOMIZE TIMER** 'ganzzahlige Zufallszahl z% erzeugen ...  
z% = INT(RND \* 6) + 1 '... zwischen 1 und 6 oder ...  
z% = INT(RND \* 90) + 10 '... zwischen 10 und 99 oder ..  
z% = INT(RND \* (max%-min%+1))+min% '... zwischen min und max (inkl.)
- Erzeugen von Zufallszahlen ohne Zahlenwiederholung: Siehe RANDOMNO.BAS

\*\*\*\*\*  
\* Schleifen und Verzweigungen  
\*\*\*\*\*

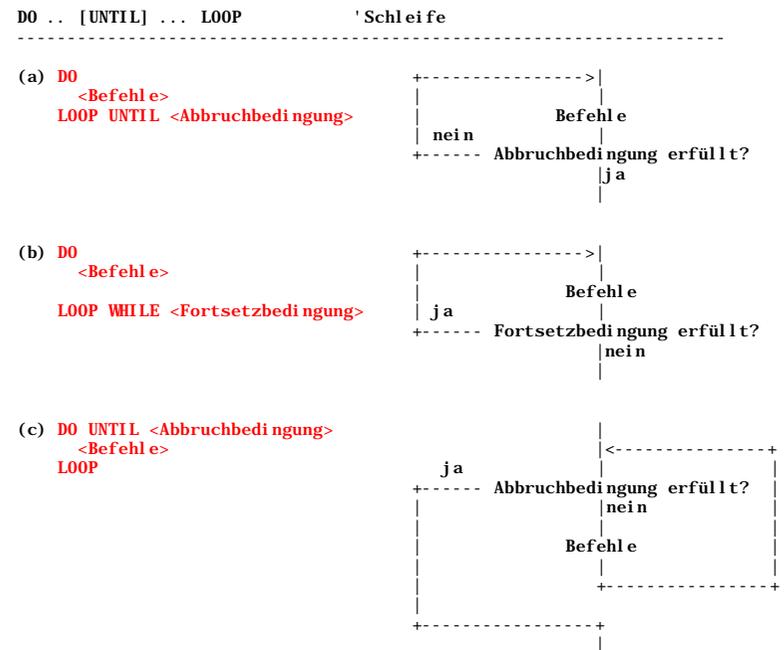
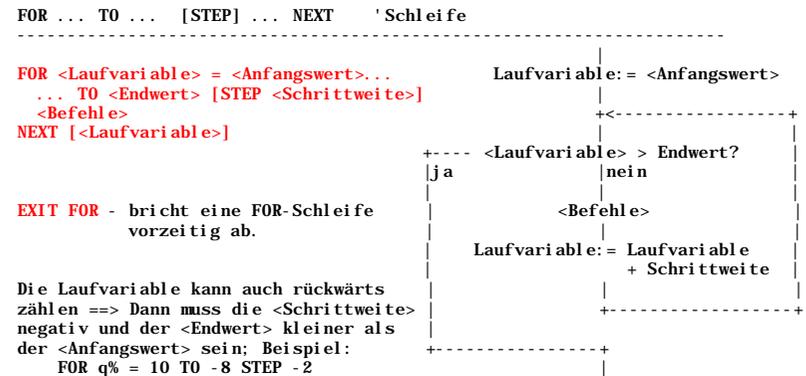
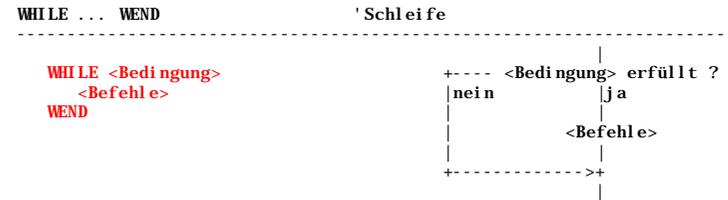
**IF ... THEN ... [ELSE]** 'Verzweigung

Anmerkung: "Bedingung" wird als erfüllt (wahr/"true") angesehen, wenn der Bedingungs-Ausdruck ungleich Null ist.

- Minimalversion in einer Zeile (bei mehrzeiligem ELSE-Block muss die Normalversion verwendet werden!):  
**IF <Bedingung> THEN <Befehl> [ELSE <Befehl>]**

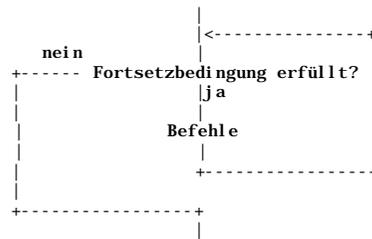


Hinweis: Bei PowerBASIC kann ein IF-Block mit **EXIT IF** verlassen werden



(d) **DO WHILE** <Fortsetzbedingung>  
<Befehle>  
**LOOP**

' ist identisch mit  
' **WHILE ... WEND** Schleife



(e) **EXIT DO** - bricht DO-Schleifen vorzeitig ab und springt zum Abbruchzweig  
(Bei PowerBASIC 'EXIT LOOP' statt 'EXIT DO' verwenden).  
Es ist z. B. auch die folgende Konstruktion möglich:

```

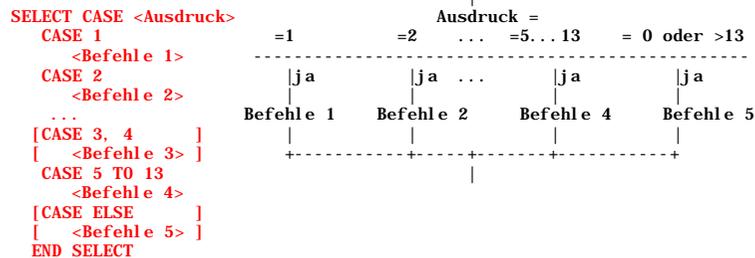
DO
  <Befehle>
  IF <Abbruchbedingung> THEN EXIT DO
  <Befehle>
LOOP
  
```

**GOTO** <Sprungmarke> - unbedingter Sprung

Ein 'unbedingter Sprung' wird bedingungslos ausgeführt. Bezüglich Sprungmarken siehe Kapitel 'Syntax'.

**SELECT CASE ... [CASE ELSE] ... END CASE** - Mehrfachverzweigung {9/51}

(<Ausdruck> kann auch vom Typ **STRING** sein, siehe Kap. Tastatureingaben)



**Kurzform:**

```

SELECT CASE i%
CASE 1: x%=2
CASE 2: x%=5
END SELECT
  
```

**Sonderfall mit CASE IS {6/112}:**

```

SELECT CASE i%
CASE IS < 23 : x=0 'i% < 23
CASE 23 TO 125: x=1 'i% = 23...125
CASE 126 : x=2 'i% = 126
CASE IS >= 127: x=3 'i% > 126
END SELECT
  
```

**Hinweise zu PowerBASIC:**

- 'CASE IS' wird von PowerBASIC nicht unterstützt.
- Ein **SELECT CASE**-Block kann bei PowerBASIC mit **EXIT SELECT** vorzeitig verlassen werden.

\*\*\*\*\*  
\* Allgemeines zu Subroutinen und Funktionen (Parameter, Lokal - /Global variablen)  
\*\*\*\*\*

**Definitionen:**

- Eine *Subroutine* ("SUB") ist ein Unterprogramm, in das man häufig benötigte Programmpassagen auslagert, die dann beliebig oft per **CALL**- oder **GOSUB**-Befehl aufgerufen werden können. Eine Subroutine gibt nach ihrer Abarbeitung die Kontrolle an das aufrufende (Haupt-)Programm zurück.
- Eine *Funktion* ("FUNCTION") ist eine Subroutine, die einen Wert zurückgibt und wie eine Variable rechts von einem Gleichheitszeichen in einer Wertzuweisung verwendet wird.
- *Prozedur* ist der Oberbegriff für Subroutinen und Funktionen. Einige (wenige) Buchautoren verwenden diesen Begriff auch gleichbedeutend mit dem Begriff "Subroutine".
- Sowohl an Subroutinen als auch an Funktionen kann man Parameter übergeben. Das sind Variablen oder Konstanten, die die SUB/FUNCTION für ihre Verarbeitungsschritte benötigt.
- *Formalparameter* sind die von der SUB/FUNCTION erwarteten Übergabeparameter.
- *Aktualparameter* sind die bei einem konkreten Aufruf tatsächlich an die SUB/FUNCTION übergebenen Parameterwerte.
- *Globalvariablen* sind Variablen, die nicht nur lokal im Hauptprogramm oder einem Unterprogramm, sondern global über Unterprogrammengrenzen hinweg zugreifbar sind. Eine *Lokalvariable* des Namens "Anna!" kann es hingegen in mehreren Unterprogrammen mit unterschiedlichen Speicherplätzen geben, und jedes Unterprogramm kann nur auf seine individuelle "Anna!" zugreifen.

Das Folgende gilt nicht für lokale SUBS (**GOSUB..RETURN**) und **FUNCTIONS** (**DEF FNx**):

- Subroutinen und Funktionen werden von QBasic in eigenen Editierfenstern bearbeitet/editiert. Dies macht ein QBasic-Programm äußerst übersichtlich, und eine SUB/FUNCTION lässt sich schnell und bequem auffinden. Die zu einer SUB/FUNCTION gehörende Befehlspassage nennt man 'SUB/FUNCTION-Definition'. Die SUB/FUNCTION-Fenster sind über <Ansicht | SUBS...> oder die F2-Taste zugreifbar. Mit den Tastenkombinationen [Umschalt + F2] und [Strg + F2] können Sie die vorhandenen SUBS/FUNCTIONs vorwärts bzw. rückwärts durchblättern.
- Neue Subroutinen und Funktionen lassen sich über <Bearbeiten | Neue SUB...> oder durch Eintippen von 'SUB <Name der SUB> [Eing.taste]' anlegen.
- Eine Subroutine/Funktion muss im aufrufenden Hauptprogramm deklariert werden. Die Deklaration wird vom QBasic-Editor automatisch wie folgt ganz am Anfang des aufrufenden Hauptprogramms eingefügt:

```

DECLARE SUB|FUNCTION <Name der Subroutine> ([<Formalparameter 1>, ...])
  
```

- Änderungen in der Parameterliste durch den Programmierer müssen in dieser Deklaration händisch nachgeführt werden.
- Anmerkung zu PowerBASIC: Die Deklaration von SUBS/FUNCTIONs im Hauptprogramm ist nur erforderlich, wenn sich die SUB/FUNCTION in einer anderen Datei befindet. Als Formalparameter sind die Variablentyp-Bezeichner statt der Parameternamen anzugeben, z. B. 'LONG' statt 'hugo&'.  
- Eine SUB | FUNCTION kann mit **'EXIT {SUB|FUNCTION}'** vorzeitig verlassen werden
- Startwert der lokalen Variablen: Alle lokalen Variablen werden bei jedem Aufruf der SUB/FUNCTION mit dem Startwert '0' (bzw. "" bei Stringvariablen) vorbesetzt. Dies gilt nicht für globale Variablen (mit **SHARED** deklariert) und wiedertrittsfähige Variablen (mit **STATIC** deklariert; siehe unten bei **STATIC** im Abschnitt 'Für Profis').
- Geltungsbereich der Variablen {9/100} {3/133} {11/122}:  
- Variablen und Felder des Hauptprogramms sind in der SUB/FUNCTION nur zugreifbar, wenn sie im Hauptprogramm als globale Variablen deklariert sind (siehe unten unter 'Variante 1') oder wenn sie als Parameter übergeben werden.  
- Lokale Variablen und Felder einer SUB/FUNCTION sind vom Hauptprogramm aus nur zugreifbar, wenn sie in der SUB/FUNCTION als globale Variablen deklariert sind (siehe unten unter 'Variante 2'). Sollen sie auch in anderen SUB/FUNCTIONs verwendet werden, so sind sie dort ebenfalls global zu deklarieren.
- Geltungsbereich der Konstanten: Alle im Hauptprogramm per **CONST** oder **DATA** deklarierten Konstanten sind auch in sämtlichen SUBs und **FUNCTIONS** zugreifbar.
- Übergabe von Feldern an SUBs und **FUNCTIONS**: Felder können als Parameter an die SUB oder **FUNCTION** übergeben werden (mit leeren Klammern ( )). Eine nochmalige Deklaration des Feldes in der SUB/FUNCTION ist nicht erforderlich. Siehe {9/98}, {6/223}, **RANDOMO.BAS** und **FLDPARAM.BAS**.

Beispiel:

```
DECLARE SUB Upro(feld()) 'Deklaration der SUB, wird von QBASIC
                          'automatisch im Hauptprogramm eingefügt
DIM feldx(3, 4)          'Aufruf der Subroutine Upro und Über-
CALL Upro(feldx()): PRINT feldx(2, 3) 'gabe eines zweidimensionalen Feldes
SUB Upro(feldy()): feldy(2, 3)=47 'Definition der SUB mit feldy als For-
END SUB                  'malparameter
```

- Übergabe von anwenderdefinierten Feldern (Verbundfeldern anwenderdefinierter Typs) an SUBS und FUNCTIONS mit 'AS ANY' {11/271}: Beispiel:  
DECLARE SUB Upro(feldx() AS ANY) 'Deklaration v. QBasic automat. eingefügt  
'Man kann auch 'AS quiz' angeben  
DIM feldx(13) AS quiz...  
CALL Upro(feldx())  
SUB Upro(feldy() AS quiz) 'AS <Typname> muss mit angegeben werden
- Über LBOUND(<Feldname>) und UBOUND(<Feldname>) kann eine SUB/FUNCTION den kleinsten und größten Index eines übergebenen Feldes ermitteln.

//////////////////////////////////// Für Profis //////////////////////////////////////

- Bei mit STATIC deklarierten lokalen Variablen und Feldern bleibt der Wert zwischen zwei Aufrufen der SUB/FUNCTION erhalten {9/98} {3/130}:
  - STATIC <Variablenname> [AS <Typ>], ... 'für Variablen
  - STATIC <Feldname> () 'für Felder
  - DIM <Feldname> (<Anzahl Feldelemente%>)
- Explizite Deklaration von Parametertypen in einer SUB/FUNCTION: Erfolgt nicht über den DIM-Befehl, sondern direkt in der Parameterliste mit 'AS <Typ>' (Ist in PowerBASIC nur mit Variablen möglich, die mit SHARED deklariert sind). Beispiel:  
DECLARE SUB Upro (anna AS LONG) 'Deklaration der SUB  
CALL Upro(otto&)... 'Aufruf der SUB  
SUB Upro (anna AS LONG): anna=anna^2 'Definition der SUB
- Parameterübergabe-Methoden 'Call by Reference' und 'Call by Value':  
siehe CALLREVA.BAS und {11/154}:
  - Call by Reference: Normalerweise werden die Parameter an eine SUB/FUNCTION 'by Reference' übergeben, d.h. die SUB/FUNCTION erhält einen Zeiger auf den Parameter, und alle Werteänderungen, die die SUB/FUNCTION an den Parametern durchführt beeinflussen den Wert der Ursprungsvariablen! Dies kann bei großen Softwareprojekten zu Softwarefehlern führen, die nur schwer zu finden sind.
  - Call by Value: Werden die einzelnen Übergabeparameter jeweils in zusätzliche Extra-Klammern gesetzt, so erhält die SUB/FUNCTION nur den Wert des Parameters, nicht seine Adresse. Bei Werteänderungen legt die SUB/FUNCTION dann eine eigene Variable an und die Ursprungsvariable bleibt unverändert. Call by Value ist nur bei der Übergabe von Einzelvariablen möglich; Felder können nur 'by Reference' übergeben werden. Konstanten werden grundsätzlich immer 'by Value' übergeben (z.B. die 5 und die 7 bei CALL MULT(5, 7))

Beispiel:

```
CALL Drehen ((wort$), (anzahl%)) 'Es wird nur der Wert von wort$
                                ' und anzahl% an die SUB übergeben ==> Die SUB
                                ' kann die Ursprungsvariablen nicht verändern
```

- Deklaration globaler Variablen und Felder (siehe {11/122} u.GLOBLVAR.BAS):  
Es gibt zwei Varianten für die Deklaration globaler Variablen und Felder:
  - VARIANTE 1: Global-Deklaration im Hauptprogramm (Normalvariante)

Variablen des Hauptprogramms, die auch in einer SUB zugänglich sein sollen, müssen im Hauptprogramm mit SHARED als Globalvariablen deklariert werden. Dies ist die am häufigsten verwendete Variante.

- Globaldeklaration (immer am Programmstart! zwei Möglichkeiten):
  - DIM SHARED {<Variable1> | <Feld> [( <Dimensionierung> )]} AS <Typ> [ , <Variable2> ... ]  
mit expliziter Typzuweisung;  
Beispiel: DIM SHARED Anna, Egon AS DOUBLE, Otto AS INTEGER  
Das Schlüsselwort DIM muss bei PowerBASIC weggelassen werden.
  - COMMON SHARED <Variable1> [ < , Variable2> , ... ] - für mit Typkennzeichen implizit deklarierte Variable, nicht für Variable, die per DIM-Befehl deklariert sind {6/178}. Felder müssen in diesem Falle mit leeren Klammern notiert werden; sie sind also dynamisch.

Beispiel: COMMON SHARED wochentag\$  
CALL WeekOfDay

```
...  
SUB WeekOfDay 'Die Variable ist auch in  
wochentag$=... 'der SUB zugreifbar
```

- Vorteile der Variante 1:
  - Alle Globalvariablen sind im Hauptprogramm in übersichtlicher Form an zentraler Stelle aufgelistet.
  - Einfacheres Handling, weniger Programmieraufwand
  - Auch statische (fest dimensionierte) Felder und anwenderdefinierte Felder mit TYPE..END TYPE sind möglich
- Nachteile der Variante 1 {11/125}:
  - Globalvariablen sind in allen SUBS/ FUNCTIONS zugreifbar, auch wenn sie dort gar nicht benötigt werden. Dies kann zum unbeabsichtigten Ändern von Variablen führen und die Fehlergefahr bei großen Softwareprojekten erhöhen.
- VARIANTE 2: Global-Deklaration in der SUB/FUNCTION (Spezialvariante)  
-----  
Variablen einer SUB/FUNCTION, die auch im Hauptprogramm zugänglich sein sollen, müssen in der SUB/FUNCTION mit SHARED als Globalvariablen deklariert werden (Umkehrung von Variante 1; nur gelegentlich verwendet)
  - Globaldeklaration:  
SHARED <Variable1> [ () ] [ AS <Typ> ] [ , <Variable2> [ () ] [ AS... ] - Globale Felder sind in diesem Falle grundsätzlich immer dynamisch, d.h. sie dürfen nicht dimensioniert werden (leere Klammern). Der Feldindex darf seltsamerweise - außer bei PowerBASIC - höchstens ca. 10 betragen, siehe GLOBLFLD.BAS).  
Sollen die Globalvariablen auch in anderen SUBS/FUNCTIONS verwendet werden, so sind sie dort ebenfalls mit SHARED zu deklarieren (ist im Hauptprogramm nicht erforderlich).
- Vorteile der Variante 2:
  - Variablen sind nur in denjenigen SUBS/FUNCTIONS bekannt, die sie auch benötigen: Dies kann in großen Softwareprojekten Fehler vermeiden helfen {11/125}.
  - Allgemein verwendbare SUBS/FUNCTIONS, die in vielen Programmen einsetzbar sind, lassen sich einfacher in ein neues Programm einfügen, da die DIM SHARED-Deklarationen im Hauptprogramm entfallen.
- Nachteile der Variante 2:
  - Sollen die globalen Variablen und Felder auch in anderen SUBS/FUNCTIONS verwendet werden, so müssen sie dort ebenfalls erneut mit SHARED deklariert werden (mehr Programmieraufwand {11/124f}).
  - Nur dynamische Felder (ohne Dimensionierung) möglich
- Rekursiver Aufruf von SUBS/ FUNCTIONS: QBasic unterstützt "Rekursion"; d.h. SUBS und FUNCTIONS können sich selbst aufrufen. Siehe {9/101}, {11/243+247}, RECURSE.BAS und SORT.BAS.

\*\*\*\*\*  
\* Subroutinen (Unterprogramme; max Länge: 64 KBytes)  
\*\*\*\*\*  
Subroutine definieren {3/131}:  
-----

- SUB <Name der Subroutine> [( <Formalparameter 1>, <Formalparameter 2> ... )]...  
... [KLARUNG]  
[ <Deklaration lokaler Variablen, Felder und Konstanten> ]  
[ <Befehle> | [EXIT SUB] 'vorzeitiger Ausprung durch EXIT SUB möglich  
END SUB  
Das Anlegen einer neuen Subroutinen-Definition erfolgt in der QBasic-Entwicklungsumgebung über den Menüpunkt "Bearbeiten | Neue SUB | <Name der Subroutine>". Die Befehlelemente "SUB <Name der Subroutine> ... END SUB" werden von QBasic in den sich öffnenden SUB-Fenster automatisch angelegt.
- Die Option STATIC erhält den Wert aller lokalen Variablen zwischen 2 Aufrufen der Subroutine {3/130}. Es lässt sich bei Bedarf auch nur ein Teil der lokalen Variablen und Felder individuell als STATIC deklarieren; siehe Beschreibung des Befehls STATIC im Kapitel 'Allgemeines zu Subroutinen..'

- Beispiel für eine Subroutine, die das Quadrat der als Parameter übergebenen Zahl bildet und anzeigt:

```

DECLARE SUB Quadrat (n%) 'Deklaration der Subroutine mit Formalparameter n%
FOR i% = 1 TO 10         'Hauptprogramm (zeigt 1^2...10^2 an)
  CALL Quadrat(i%)      'Aufruf der Subroutine mit Aktualparameter i%
NEXT i%
END
SUB Quadrat (n%)        'Definition der Subroutine
  n2% = n% ^ 2          '(wird in extra Fenster angezeigt)
  PRINT n%, n2%
END SUB

```

#### Subroutine aufrufen

Es gibt zwei Methoden zum Aufruf einer Subroutine: Mit CALL und ohne CALL.

- Aufruf einer SUB mit CALL:  
**CALL <Name der Subroutine> [(**<Aktualparameter 1>**, **<Aktualparameter 2>** ...)]**  
 - Aufruf einer SUB ohne CALL  
**<Name der Subroutine> [(**<Aktualparameter 1>**, ...)]**

Die Klammern um die Parameterliste wird bei dieser Methode weggelassen.

Beispiel: upro otto& 'statt CALL upro(otto&)

Diese Aufruf-Syntax wird von Profis gern verwendet. Ich persönlich halte den SUB-Aufruf mit CALL für übersichtlicher. PowerBASIC kennt den SUB-Aufruf ohne CALL übrigens nicht.

#### Subroutine im aufrufenden Hauptprogramm deklarieren:

... und Geltungsbereich d. Variablen: Siehe Kapitel Allgemeines zu Subroutinen...

\*\*\*\*\*  
 \* Funktionen (Unterprogramme mit Rückgabewert); max Länge 58 KB {3/124}  
 \*\*\*\*\*

#### Funktion definieren:

- **FUNCTION <Name d. Funktion[Typkennzeichen]> [(**<Formalparam 1>**, **<Formalparam 2>** ...)]**  
 ... [STATIC]  
 [**<Deklaration lokaler Variablen, Felder und Konstanten>**]  
**<Befehle> | EXIT FUNCTION** 'vorzeitiger Aussprung mit EXIT FUNCTION möglich  
**<Name d. Funktion> = <Ausdruck>** 'Rückgabewert zuweisen; dies muss unbedingt  
 direkt vor END FUNCTION erfolgen; notfalls  
 'Zwischenvariable einführen  
**END FUNCTION**

- Die Option STATIC erhält den Wert aller lokalen Variablen zwischen 2 Aufrufen der Funktion {3/130}. Es lässt sich bei Bedarf auch nur ein Teil der lokalen Variablen und Felder individuell als STATIC deklarieren; siehe Beschreibung des Befehls STATIC im Kapitel 'Allgemeines zu Subroutinen...'.  
 - Beispiel einer FUNCTION vom Typ INTEGER, die das Quadrat der als Parameter übergebenen Zahl bildet und zurückliefert:  

```

DECLARE FUNCTION Quadrat% (n%) 'Deklaration d. Funktion m. Formalparameter n%
FOR i% = 1 TO 10              'Hauptprogramm (zeigt 1^2...10^2 an)
  PRINT i%, Quadrat%(i%)     'Aufruf der Funktion mit Aktualparameter i%
NEXT i%
END
FUNCTION Quadrat% (n%)       'Definition der Funktion
  Quadrat% = n% ^ 2          '(wird in extra Fenster angezeigt)
END FUNCTION

```

#### Funktion aufrufen:

**x=<Name d. Funktion[Typkennzeichen]> [(**<Aktualparameter 1>**, **<Aktualparameter 2>**...)]**

Der Aufruf darf nur in einer Wertzuweisung (rechts von einem Gleichheitszeichen) oder in einem Ausdruck stehen (z. B. hinter einer PRINT-Anweisung).

Beispiele für den Aufruf der Funktion otto\$ mit Übergabeparameter 5:

- anna\$ = otto\$(5)  
 - PRINT otto\$(5)  
 - anna\$ = otto\$(5) +; "ist lieb"

#### Funktion im aufrufenden Hauptprogramm deklarieren:

... und Geltungsbereich d. Variablen: siehe Allgemeines zu Subroutinen...

#### Funktionen mit mehr als einem Rückgabewert :

Die Variablen für die Rückgabewerte werden ebenfalls als Aktual- bzw. Formalparameter in die Parameterliste eingetragen - wie die Übergabeparameter. Diese Vorgehensweise gilt auch für Subroutinen. Siehe auch MEHRUECK.BAS sowie {9/94+96} und {6/182}.

\*\*\*\*\*  
 \* Lokale Subroutinen (GOSUB)  
 \*\*\*\*\*

#### - Hinweise:

- Die Verwendung lokaler Subroutinen wird normalerweise nicht empfohlen; sie dienen weitgehend nur der Kompatibilität zu BASICA und GW-BASIC.
- Die Definition einer lokalen Subroutine ist auch innerhalb einer SUB möglich und hat den Vorteil, dass die aufrufende SUB als zusammenhängender Textblock einfacher in andere QBasic-Programme kopierbar ist.
- Im Hauptprogramm sollte die lokale Subroutine hinter dem Programmende nach dem END-Befehl definiert werden.
- Eine Parameterübergabe an eine lokale Subroutine ist nicht möglich
- Deklaration : nicht erforderlich
- Definitionsbeispiel: Potenz: 'Übergabeparameter nicht möglich !  
 $x = 2 ^ i$   
**RETURN**
- Aufrufbeispiel : **GOSUB** Potenz
- Variablen : Eine lokale Subroutine hat keine lokalen Variablen, sondern kennt alle Variablen des aufrufenden Programms und umgekehrt

#### - Beispiel einer lokalen Subroutine 'Quadrat' die das Quadrat d. Zahl x anzeigt:

```

FOR i = 1 to 10 'Hauptprogramm (zeigt 1^2...10^2 an)
  GOSUB Quadrat
NEXT i
END 'Ende des Hauptprogramms
Quadrat: 'lokale Subroutine (Sprungmarke)
  print i^2
RETURN 'Rückkehr zum Hauptprogramm

```

\*\*\*\*\*  
 \* Lokale Funktionen (DEF FN...) {9/88}  
 \*\*\*\*\*

#### - Hinweise:

- Die Verwendung lokaler Funktionen wird normalerweise nicht empfohlen; sie dienen weitgehend nur der Kompatibilität zu den älteren BASIC-Dialekten BASICA und GW-BASIC.
- Eine lokale Funktion muss am Beginn des Hauptprogramms definiert werden. Eine Definition am Ende des Hauptprogramms und in SUBS/FUNCTIONs ist - im Gegensatz zur lokalen Subroutine - nicht möglich.
- Eine lokale Funktion kann über EXIT DEF vorzeitig verlassen werden.
- An eine lokale Funktion lassen sich beliebig viele Parameter übergeben.
- Definitionsbeispiel 1: **DEF FNpotenz! (basis!) = 2 ^ basis!** 'einzeilige Funktion  
 'Name muss mit 'FN' beginnen;
- Definitionsbeispiel 2: **DEF FNpotenz! (basis!) 'Mehrzeilige Funktion  
 <Befehle> | EXIT DEF 'Name muss mit FN beginnen  
 FNpotenz! = 2 ^ basis!** 'vorzeitiger Aussprung mit  
**END DEF** 'EXIT DEF möglich
- Aufrufbeispiel : **PRINT FNpotenz! (x)** 'Vor dem Aufruf muss die Definition erfolgt sein!
- Variablen : Eine lokale Funktion hat keine lokalen Variablen, sondern kennt alle Variablen des aufrufenden Programms und umgekehrt
- Beispiel einer lokalen Funktion 'FNquadrat' die das Quadrat der Zahl x bildet:  

```

DEF FNquadrat# (n) 'Lokale Funktion vom Typ DOUBLE
  FNquadrat# = n ^ 2
END DEF
FOR i = 1 TO 10 'Hauptprogramm (zeigt 1^2...10^2 an)
  PRINT FNquadrat#(i)
NEXT i

```

```

*****
* DOS-Befehl oder externes EXE-Programm/ BAT-Batchdatei aufrufen
*****
- SHELL ["<Anweisung>"] - gibt den String "<Anweisung>" am DOS-Prompt aus.
  Nach Beenden des DOS-Programms erfolgt ein Rücksprung zum QBasic-Programm.
  Fehlt die 'Anweisung', so wird zum DOS-Betriebssystem gewechselt und bei
  Eingabe von "EXIT" erfolgt der Rücksprung zum QBasic-Programm.
- Beispiele:
  - SHELL "calcul.exe" 'externes EXE-Programm aufrufen
  - SHELL "dir c:\\" 'Liste aller Ordner und Dateien des Stammverzeichnis-
    nisses in Laufwerk C: anzeigen
  - SHELL "dir c:\\" >tmp.txt 'Verzeichnisliste von Laufwerk C: in die Text-
    'datei tmp.txt eintragen (> bewirkt eine
    'Ausgabeumleitung')
  - SHELL "COPY "+ Adatei$ + " " + Bdatei$ 'Adatei nach Bdatei kopieren
  - SHELL "cd >xx.txt" 'aktuellen Pfadnamen in Datei xx.txt schreiben
    '(> bewirkt eine 'Ausgabeumleitung')
  - SHELL "echo. | date" 'Datum anzeigen mit automatischer Betätigung der
    'Eingabetaste, um das DOS-Kommando DATE zu beenden
  - SHELL "echo j | del c:\temp\*. *" 'Alle Dateien im Verzeichnis c:\temp
    'löschen mit automatischer Betätigung durch "j |"
    '("Eingabeumleitung zur Pipe")

```

#### BASIC-DOS-Dateisystembefehle aufrufen

```

-----
- Hinweise:
  - Platzhalter "*", "?" in den Pfadnamen sind erlaubt ('Wildcards').
  - Die Fehlerbearbeitung, z.B. bei nicht vorhandenen Dateien, muss von Hand
    ausprogrammiert werden; siehe Abschnitt Fehlerbehandlung im Kapitel
    'Dateibearbeitung - Allgemeines'. Daher ist der DOS-Befehlsaufruf über
    SHELL häufig günstiger.
- CHDIR <Pfadname$> - Wechsel in ein anderes Verzeichnis
- KILL <[Pfadname$]Dateiname$> - Datei löschen
- MKDIR <Pfadname$> - ein neues Unterverzeichnis erstellen
- RMDIR <Pfadname$> - ein Unterverzeichnis löschen
- NAME <alter Name$> AS <neuer Name$> - Datei oder Verzeichnis umbenennen
- FILES <[Pfadname$]> - zeigt den Inhalt des aktuellen Verzeichnisses oder
  eines angegebenen Pfades im aktiven Laufwerk an,
  ähnlich dem DOS-Befehl DIR.
- DIR$ [( <[Pfadname$]> <Dateiname$> ) ] - Liefert den Namen der in einem
  Verzeichnis vorhandenen Dateien zurück (sehr
  komfortabel; nur ab QuickBASIC 7.1/PDS)

```

//////////////////////////////////// Für Profis //////////////////////////////////////

```

- DOS-Umgebungsvariable lesen und ändern {11/468}: Die in der AUTOEXEC.BAT
  gesetzten Umgebungsvariablen lassen sich durch ein QBasic-Programm auslesen
  und ändern. Änderungen bleiben jedoch nur während der Laufzeit des
  QBasic-Programms gültig:
  - ENVIRON$ (<Nummer%>) - liefert den Setzbefehl der n-ten momentan
    gesetzten Umgebungsvariablen als Zeichenkette zurück.
    Beispiel: FOR i = 1 TO 20: PRINT i; ENVIRON$(i); NEXT 'Anzeige der
    'ersten 20 Umgebungsvariablen (mehr gibt es meist nicht)
  - ENVIRON$ (<NameDerUmgebungsvariablen$>) - liefert den in der AUTOEXEC.
    BAT stehenden Wert für die in Großbuchstaben angegebene Umgebungs-
    variable als Zeichenkette zurück.
    Beispiele: PRINT ENVIRON$("PATH"), ENVIRON$("PROMPT")
    PRINT ENVIRON$("BLASTER")
  - ENVIRON (<Name DerUmgebungsvariablen$> "=" <Setzwert$> | <">) - Um-
    gebungsvariable setzen/löschen.
    Beispiele: ENVIRON "PATH= C:\PROGIS" 'Pfad-Umgebungsvariable ändern
    ENVIRON "PATH=" 'Pfad-Umgebungsvariable löschen

```

\*\*\*\*\*  
 \* Modulare Programmierung und Bibliotheken (CHAIN, \*.LIB)  
 \*\*\*\*\*

Modulare Programmierung heißt, dass der Programmierer sein Programm auf mehrere  
 Dateien aufteilt, welche dann bei anderen Softwareprojekten bei Bedarf wieder-  
 verwendbar sind.

Bei QBasic lassen sich mehrere Programmdateien mit CHAIN und RUN miteinander  
 verketten. Bei QuickBASIC und PowerBASIC gibt es mehr Möglichkeiten der

modularen Programmierung; die wichtigsten davon sind die Verwendung von  
 Bibliotheken (Libraries), MAK-Modulen und Include-Dateien.

```

- CHAIN [Pfadname$] <Dateiname$> - übergibt die Kontrolle von dem aktu-
  ellen Programm an das BASIC-Programmmodul <Dateiname$> {11/454}.
  Eine automatische Rückkehr ins alte Programm findet nicht statt.
  Beispiele: CHAIN "C:\DOS\TEST.BAS"
             CHAIN "prog2.exe"
  Mit CHAIN kann man die Speichergrenze von 64 KB je Quellsprache-
  programm umgehen, indem man sein Programm auf mehrere Dateien
  aufteilt.
- CHAIN <Zeilennummer> - Diese Befehlsvariante startet das laufende Programm
  neu, setzt alle Variablen zurück und springt die angegebene
  Zeilennummer an. Der Start eines externen Programms ist hiermit
  nicht möglich.
- COMMON [SHARED] <Variablenliste> - Definition von Variablen und Feldern, die
  auch von anderen externen "gechainten" Programmmodulen verwendbar
  sind. Die Reihenfolge der Variablen in der Variablenliste muss in
  beiden Programmen genau gleich sein! Bei Verwendung von SHARED sind
  die Variablen auch von allen SUBS/FUNCTIONS zugreifbar. {11/454}
  Felder müssen erst deklariert und dann dimensioniert werden.
  Beispiel: COMMON [SHARED] fel d%( )
             DIM fel d%(199)
- RUN [Pfadname$] <Dateiname$> - startet ein externes Programm. Wie
  CHAIN, jedoch werden vor dem Start des externen Programms alle
  Variablen gelöscht und alle offenen Dateien geschlossen.
  Beispiel: RUN "C:\DOS\TEST.BAS"
- In QuickBASIC lassen sich über den QuickBASIC-Linker Programme aus mehreren
  Dateien und Bibliotheken (*.LIB und *.QLB) zusammenbinden. In
  Bibliotheken lagert man häufig benutzte SUBS und FUNCTIONS aus.
- In PowerBASIC lassen sich über die Compiler-Anweisungen (Direktiven) SLIB und
  SLINK Bibliotheken sowie externe Programmdateien/ Units einbinden.

```

\*\*\*\*\*  
 \* Datei bearbeitung - Allgemeines, Dateierarten, Fehlerbehandlung {11/285ff}  
 \*\*\*\*\*

#### Allgemeines zur Dateibearbeitung:

```

=====
- Der Umgang mit Dateien ist in der einschlägigen Literatur und in der QBasic-
  Onlinehilfe nur bruchstückhaft und oft recht unsystematisch dargestellt.
  Viele Details findet man nur durch Probieren heraus. Diesem Mangel möchte
  das QBasic-Kochbuch abhelfen. Es behandelt daher die Dateizugriffe umfassend
  und mit allen Details.
- Es lassen sich max. 255 Dateien beliebiger Größe mit jeweils max.
  2 147 483 647 Datensätzen à max. 32 KB in einem Programm bearbeiten
- Jede offene Datei ist durch eine #Dateinummer% (#1...#255) gekennzeichnet.
- Maximal 16 Dateien dürfen gleichzeitig geöffnet sein.
- I/O-Geräte sind ebenfalls als Dateien definiert, z.B. "LPT1:" = Drucker,
  "SCRN:" = Monitor, "COM1:" = 1. serielle Schnittstelle/Maus, "KEYBD:" = Tasta-
  tur usw. {9/76} {11/304}.
- So kann man prüfen, ob eine Datei vorhanden ist:
  Variante 1: Bei sequentiellen Dateien, die zum Lesen geöffnet werden, Fehler
  "Datei nicht gefunden" abfragen entsprechend dem folgenden Bei-
  spiel: ON ERROR GOTO fehler
  INPUT "Welche Datei wollen Sie oeffnen "; dateiname$
  OPEN dateiname$ FOR INPUT AS #1
  GOTO weiter
  fehler: PRINT "Datei nicht vorhanden!"
  INPUT "Geben Sie d.korrekten Dateinamen ein"; dateiname$
  RESUME 'Ruecksprung zum fehlerverursachenden Befehl
  weiter: PRINT "Datei ist vorhanden"
  '... Hier folgen die weiteren Befehle
  Siehe auch SEQERROR.BAS.
  Variante 2: Bei allen anderen Dateierarten: Dateilänge auf "0" abfragen:
  IF LOF (<Dateinr. ohne #>) > 0 THEN ... 'Dateilänge > 0 ?
  Funktioniert bei Direktzugriffs-Dateien und binären Dateien. Bei
  sequentiellen Dateien nur in der Zugriffsart OUTPUT verwendbar.
  Variante 3: Prüfen, ob ein definiertes Datum rücklesbar ist, siehe
  DATEIVOR.BAS (bei Direktzugriffs-Dateien). Wenn "0" ausge-
  lesen wird, ist die Datei nicht vorhanden.
  Welche Variante sinnvoll ist, hängt von der Dateierart und der Zugriffsart ab.

```

//////////////////////////////////// Für Profis //////////////////////////////////////

- Weitere Varianten der unten angegebenen OPEN-Befehle zum Öffnen einer Datei findet man in der QBasic-Onlinehilfe unter OPEN und in {11/303+311}, z. B. **ACCESS READ WRITE**: Öffnen zum Lesen und zum Schreiben
- Freigabe und Sperren von Dateien im Netzwerk: Siehe QBasic-Online-Hilfe unter **OPEN, ACCESS, LOCK, UNLOCK sowie SHARED** {11/303}
- **FREEFILE** - Die Funktion FREEFILE liefert die nächste noch freie, unbenutzte Dateinummer zurück {11/466}. FREEFILE kann von SUBS/FUNCTIONs genutzt werden, die nicht wissen können, welche und wieviele Dateien bereits vom Hauptprogramm geöffnet sind.
- **CLOSE** (ohne Parameter) oder **RESET** schließt alle offenen Dateien.
- **WIDTH** (<#Dateinummer%>), <SpaltenZahl%> - legt die Spaltenzahl (Zeilenlänge) in einer Datei fest (wenig gebräuchlich {11/464})
- **FILEATTR** (<Dateinr>, {1|2}) - liefert bei Attribut =1 den aktuellen Zugriffsmodus auf die Datei zurück (1=INPUT, 2=OUTPUT, 4=RANDOM, 8=APPEND, 32=BINARY) und liefert bei Attribut=2 die - wenig interessante - DOS-Dateinr. zurück {11/467}.
- Mit **BSAVE/BLOAD** lassen sich Daten zwischen einem absolut adressierten Speicherbereich (z. B. Bildschirmspeicher) und einer Datei transferieren; siehe Abschnitt 'Speicherbereich mit BSAVE/BLOAD in Datei schreiben...' im Kapitel 'Direkter Speicherzugriff...'

Arten von Dateien und Kriterien zur Auswahl der richtigen Dateiart:

=====

Es gibt die folgenden 4 Dateiarten:

- Sequentielle Dateien (gebräuchlichste Dateiart)
- Direktzugriffs-Dateien mit TYPE-Puffer (häufig verwendet)
- Direktzugriffs-Datei mit FIELD-Puffer (weniger gebräuchlich)
- Binäre Dateien (weniger gebräuchlich)

Diese Dateiarten unterscheiden sich bezüglich der Datenorganisation und der Zugriffsmechanismen. Bei allen Dateiarten außer der Binären Datei wird der Inhalt einer Datei gedanklich in Datensätze unterteilt. Beim Zugriff auf die Datei wird grundsätzlich immer ein ganzer Datensatz gelesen oder geschrieben. Ein Datensatz ist eine geordnete Ansammlung von Teil-Informationen, die in 'Feldern' hinterlegt sind (nicht zu verwechseln mit den im Kapitel 'Felder' beschriebenen indizierten Feldern). Ein Datensatz kann z. B. alle zu einer Person gehörenden Daten in einer Adressdatenbank sein und Felder für "Nachname", "Vorname", "Adresse" und "Telefonnummer" enthalten.

Die Dateiarten werden im Folgenden kurz mit ihren Vor- und Nachteilen sowie ihren typischen Anwendungsschwerpunkten vorgestellt. Anschließend ist jede Dateiart in einem eigenen Kapitel im Detail beschrieben.

Sequentielle Dateien

-----

- Sequentielle Dateien dienen zur Speicherung von "Datensätzen", die grundsätzlich aus Textstrings (ASCII-Zeichen) bestehen. Numerische Werte werden ebenfalls als Strings abgelegt, ähnlich wie bei der Bildschirmausgabe über PRINT {6/336}.
- Bei einer Sequentiellen Datei entspricht ein Datensatz immer einer Textzeile. Er kann beliebig lang sein und endet mit Enter und Zeilenvorschub (CR + LF = CHR\$(13) + CHR\$(10) = Carriage Return + Linefeed)
- Ein Datensatz kann beliebig viele Felder enthalten, die beliebig lang sein können und durch Kommas voneinander getrennt sind. Die Datensätze können auch unterschiedliche Anzahl von Feldern beinhalten.
- Ein direkter wahrfreier Zugriff auf einen beliebigen Datensatz ist nicht möglich. Wie der Name "sequentiell" (lat. "in einer Folge") schon sagt, lassen sich die Datensätze nur in einer lückenlosen Folge beginnend beim ersten Datensatz aus der Datei auslesen und in die Datei hineinschreiben - beginnend am Dateianfang. Will man einen Datensatz mitten in der Datei lesen, schreiben oder einfügen, muss man erst alle vorhergehenden Datensätze lesen.
- Das Dateiformat einer Sequentiellen Datei ist fast identisch mit dem CSV-Format (Comma-Separated Variables), das von fast allen Datenbankprogrammen (MS Access, dBase usw.) und Tabellenkalkulationen als Im- und Export-Format unterstützt wird. Daher lassen sich hiermit erzeugte Datenbanken und Tabellen in der Regel leicht mit QBasic weiterverarbeiten.
- Vorteile : - Einfaches Handling, gebräuchlichste Dateiart neben der Direktzugriffs-Datei mit TYPE-Puffer
  - Datensätze können unterschiedlich lang und unterschiedlich strukturiert sein (spart u. U. viel Speicherplatz)
  - Ideal für kleine Dateien, die leicht in den Arbeitsspeicher

hineinpassen, z. B. INI-Dateien und Highscore-Listen. Bei Highscore-Listen ist jedoch wegen der textbasierten Darstellung eine Manipulation durch Unbefugte mit einem beliebigen Editor leicht möglich!

- Nachteile: - Es sind nur Texte speicherbar, numerische Größen werden ebenfalls als Text abgelegt und benötigen daher mehr Speicherplatz, z. B. INTEGER-Wert 4711 ==> "4711" (4 statt 2 Bytes).
- Es ist kein wahrfreier Direktzugriff auf Datensätze mitten in der Datei möglich. Es sind z. B. 25 Lesezugriffe erforderlich, um auf den 25. Datensatz einer Datei zuzugreifen!
- Ist ein direkter Zugriff auf beliebige Datensätze gewünscht, ohne erst die vorausfolgenden lesen/schreiben zu müssen, so muss die Datei zunächst 'am Stück' in ein RAM-Feld eingelesen und nach der Bearbeitung komplett wieder in die Datei geschrieben werden. Als Alternative hierzu kann man bei großen Dateien vorübergehend eine "Zwischendatei" als Puffer verwenden.
- Die Anzahl der in einer Datei gespeicherten Datensätze ist unbekannt.
- Die genaue Position eines bestimmten Datensatzes innerhalb der Datei ist unbekannt.

Direktzugriffs-Dateien mit TYPE-Puffer

-----

- Direktzugriffs-Dateien, oft auch "Random-Dateien" genannt (random = beliebig), enthalten Datensätze fester Länge, auf die beliebig über eine Datensatznummer zugegriffen werden kann.
- Ein Datensatz kann aus beliebig vielen Feldern beliebigen Datentyps, jedoch fester Länge bestehen. Alle Datensätze müssen gleich viele und gleich lange Felder haben. Ein bestimmtes Feld muss in allen Datensätzen immer wieder denselben Datentyp haben.
- Als Schreib- / Lese-Puffer wird normalerweise ein mehrdimensionales Feld mit anwenderspezifischem Typ (TYPE...END TYPE) verwendet (siehe Kapitel 'Felder' unter ...Verbundfeld...). Diesen Puffer nennen wir hier 'TYPE-Puffer'.
- Vorteile : - Es muss nur der gerade benötigte Datensatz in den RAM-Speicher gelesen werden. Die restlichen Datensätze können in der Datei verbleiben - ideal für große Dateien.
  - Bequemer Zugriff auf beliebige Datensätze über Datensatznummer
  - Flexible Datensatzstruktur mit Teil-Feldern beliebigen Typs
  - neben der sequentiellen Datei die gebräuchlichste Dateiart, gut geeignet für die Bearbeitung einer Datenbank mit fester Struktur
- Nachteile: - Datensätze und deren Teilfelder haben im Gegensatz zur sequentiellen Datei immer eine konstante Länge ==> u. U. hoher Speicherplatzbedarf.
  - Weniger geeignet für Datenbankprogramme, mit denen viele unterschiedliche Datenbank-Dateien angelegt und hantiert werden sollen. Hierfür ist die Dateiart mit FIELD-Puffer besser geeignet.
  - Wird von früheren PowerBASIC-Versionen nicht unterstützt (diese kennen keine TYPE...END TYPE Deklaration; bei V3.5 jedoch vorhanden).

Direktzugriffs-Dateien mit FIELD-Puffer (weniger gebräuchlich)

-----

- Die Zugriffstechniken sind nahezu identisch mit den vorgenannten Dateien mit TYPE-Puffer. Jedoch erfolgt das Schreiben/Lesen nicht über einen anwenderspezifischen Datentyp (mehrdimensionales Feld), sondern über einen speziellen FIELD-Puffer, der nur einen Datensatz aufnehmen kann. Ein FIELD-Puffer wird mit dem FIELD-Befehl angelegt (field, engl. = Feld).
- Der FIELD-Puffer enthält ausschließlich Strings und kann in beliebig viele Teilfelder unterteilt sein. Numerische Werte müssen mit speziellen QBasic-Befehlen in 'Pseudostrings' umgewandelt werden
- Die Datensatzlänge und damit die Länge des FIELD-Puffers für eine konkrete Datei muss grundsätzlich immer gleich lang sein.
- Dieser Dateityp wird laut Microsoft weitgehend nur zur Kompatibilität mit den alten BASIC-Sprachen BASICA und GW-BASIC noch unterstützt. Für neue Softwareprojekte empfiehlt Microsoft die Verwendung eines TYPE-Puffers. Trotzdem behandeln viele Q(uick)Basic-Bücher leider nur Direktzugriffs-Dateien mit FIELD-Puffer und nicht die viel einfachere zu verwendenden Dateien mit TYPE-Puffer. Wahrscheinlich liegt das daran, dass GW-BASIC und ältere QuickBASIC Versionen die Variante mit TYPE-Puffer nicht unterstützen.
- Vorteile gegenüber Dateien mit TYPE-Puffer:

- Die Datenstruktur des FIELD-Puffers ist auch noch zur Laufzeit beliebig änderbar. Daher ist dieser Dateityp ideal geeignet zur Entwicklung regelrechter Datenbankprogramme, die unterschiedlichste Datenbankdateien anlegen und bearbeiten können {9/139ff}.
  - Auch bei älteren PowerBASIC-Versionen < V3.5 verwendbar.
- Nachteile gegenüber Dateien mit TYPE-Puffer:
- Etwas umständliche Handlung
  - Numerische Werte müssen vor dem Schreiben in "Pseudostrings" umgewandelt und nach dem Auslesen aus der Datei wieder entsprechend in numerische Werte rückgewandelt werden.

#### Binäre Dateien (weniger gebräuchlich)

- Binäre Dateien sind quasi Byte-Felder ohne besondere Datensatzstruktur. Die Datenzugriffe erfolgen streng byteweise, d.h. es werden immer 8 Bits gemeinsam gelesen oder geschrieben. Der Zugriff erfolgt dabei über die Angabe der Byteposition an beliebiger Stelle oder fortlaufend.
- Vorteile : - Flexibelste Dateiart
- Nachteile: - Der Entwickler muss sämtliche Strukturen, Dateizeiger (beinhaltet die aktuelle Byteposition) und Dateninterpretationen selbst ausprogrammieren.
- Da QBasic keinen Datentyp BYTE kennt, ist man häufig gezwungen, Tricks zu verwenden wie z.B. 'Pseudostrings' bestehend aus nur einem Zeichen.

#### Fehlerbehandlung {3/181} {9/127} {6/358} {11/351}:

- Die Fehlercodes findet man in {9/135} und in der QBasic-Hilfe unter <Hilfe | Inhalt | Laufzeit-Fehlercodes> aufgelistet.
- Fehlerroutine aufrufen, die unter <Marke2S> definiert ist (siehe DRIVECHK.BAS):
 

```
ON ERROR GOTO <Marke2S> 'muss vor dem Fehler-verursachenden Befehl stehen
<Marke1S>:          .... 'hier kommen die Fehler-verursachenden Befehle
<Marke2S>:          'Beispiele für Fehlerbearbeitungen:
IF ERR = 11/7 THEN.. 'Division durch 0 | zuwenig Speicherplatz
IF ERR = 53 THEN ... 'Datei nicht gefunden
IF ERR = 61/72 THEN.. 'Disk voll | defekt
IF ERR = 64/76 THEN.. 'Dateiname unzulässig | Pfad nicht gefunden
IF ERR = 71 THEN ... 'Festplatte/ Diskette nicht bereit
[RESUME |          'an der Zeile mit der Fehlerstelle fortsetzen
RESUME NEXT |     'an der Zeile hinter der Fehlerstelle fortsetzen
RESUME <Marke1S>] 'hinter Marke1S fortsetzen
```
- Fehler simulieren (für Testzwecke): **ERROR <fehlernr%>** {11/357}
- Auslösung anwenderdefinierter Fehler (Fehlernummern über 100; weniger gebräuchlich): z.B. **IF zahl% = 1 THEN ERROR 111** {11/358}
- **ERDEV** - liefert den DOS-Fehlercode zurück {11/358}
- **ERDEVS** - liefert den Namen des Fehler-verursachenden Geräts zurück, z.B. "A:" bei Fehler des Diskettenlaufwerks {11/358}
- **ERL** - liefert die Nummer der Fehler-verursachenden Programmzeile zurück, falls die Zeilen im Programm nummeriert sind, sonst '0' {11/359}
- **ON ERROR GOTO 0** - Fehlerkontrolle abschalten (funktioniert nicht bei schweren Fehlern wie Division durch '0') {11/359}

#### \* Sequentielle Dateien {9/109} {3/145} {6/290}

- Allg. Hinweise:
  - Siehe auch Kapitel 'Dateibearbeitung - Allgemeines..' und FILE-SEQ.BAS
  - In einer sequentiellen Datei lassen sich Werte beliebigen Datentyps ablegen. Text wird immer in Anführungszeichen abgespeichert, außer beim Schreiben der Datensätze mit dem PRINT-Befehl {11/305}. Numerische Werte werden automatisch als abdruckbare ASCII-Zeichen abgespeichert.
- Beispiel: 

```
OPEN "XXX" FOR OUTPUT AS #1
TS = "Anna": x%=4711
WRITE #1, TS, x%
```

==> Die Datei xxx enthält den folgenden Text:  
 "ANNA",4711 <CR+LF>. Man beachte, dass der String 'ANNA' in Anführungszeichen in der Datei abgelegt ist (belegt 6 statt 4 Zeichen). Die Zahl 4711 wird als Text mit 4 Zeichen hinterlegt.  
 <CR+LF> = Enter und Zeilenvorschub = CHR\$(13) + CHR\$(10).

- Die Anzahl der in einer sequentiellen gespeicherten Datensätze ist prinzipiell unbekannt. Sie muss gegebenenfalls als Zahlenwert in einer zweiten Datei abgelegt werden.
- Zugriffsarten: Der Dateizugriff ist über die folgenden Zugriffsarten möglich:
  - **OUTPUT** ==> Datensätze werden ab Dateibeginn fortlaufend geschrieben (mit WRITE, PRINT oder PRINT USING). VORSICHT: Der alte Dateinhalte wird beim Öffnen einer Datei in der Zugriffsart OUTPUT gelöscht!!!
  - **APPEND** ==> Datensätze werden hinten angefügt (alter Dateinhalte bleibt beim Öffnen erhalten und wird nicht gelöscht).
  - **INPUT** ==> Datensätze werden ab Dateibeginn gelesen (mit INPUT oder LINE INPUT)
- **OPEN [Pfadname\$] <Dateiname\$> FOR {OUTPUT|INPUT|APPEND} AS #<Dateinr.1...255>**
  - Öffnen einer Datei in einer der oben genannten Zugriffsarten.
  - Bei OUTPUT und INPUT wird der Dateizeiger auf den ersten Datensatz gesetzt.
  - Bei APPEND wird der Dateizeiger hinter den letzten in der Datei gespeicherten Datensatz gesetzt.
  - Beim Öffnen einer nicht vorhandenen Datei in der Zugriffsart INPUT wird das Programm mit Fehler abgebrochen. Dies kann mit ON ERROR GOTO... abgefangen werden; siehe 'Fehlerbehandlung' im Kapitel 'Dateibearbeitung Allgemeines...'
- **CLOSE [#<Dateinr.>]**
  - Schließen einer Datei; vor einem Wechsel der Zugriffsart (durch OPEN...FOR) muss die Datei wieder geschlossen werden.
  - Bei weggelassener Dateinr. werden alle offenen Dateien geschlossen.
- Schreiben und Lesen von strukturierten Datensätzen (u.U. mit Teilfeldern):
  - **WRITE #<Dateinr.>, <Variable1> [, <Variable2>... , <Variable n>]**
    - Schreiben eines Datensatzes [bestehend aus mehreren Teilfeldern] aus Variable(n) in eine Datei. Zwischen den Teilfeldern werden Kommas, hinter dem Datensatz ein <CR+LF> eingefügt. Strings werden in "Anführungszeichen" abgelegt
    - Die Daten werden an der aktuellen Dateizeigerposition in die Datei eingefügt {6/298}. Der Dateizeiger wird anschließend inkrementiert, d.h. auf den nächsten zu lesenden Datensatz gesetzt.
    - Die Datei muss vorher einmal in der Zugriffsart OUTPUT oder APPEND geöffnet worden sein.
  - **INPUT #<Dateinr.>, <Variable1> [, <Variable2>... , <Variable n>]**
    - Lesen eines Datensatzes [bestehend aus mehreren durch Kommas getrennten Teilfeldern] aus einer Datei in die Variable(n).
    - Ist nur Variable1 angegeben, so erfolgt das Lesen nur bis zum ersten Komma.
    - Die Daten werden an der aktuellen Dateizeigerposition aus der Datei gelesen {6/298}. Der Dateizeiger wird anschließend inkrementiert, d.h. auf den nächsten zu lesenden Datensatz gesetzt.
    - Die Datei muss vorher einmal in der Zugriffsart INPUT geöffnet worden sein (mit OPEN ... FOR INPUT ...).
- Schreiben und Lesen von unstrukturierten, nicht in Teilfelder unterteilten Strings, die auch Kommas enthalten können, z.B. normale Textdateien (weniger gebräuchliche Alternative zu WRITE und INPUT):
  - **PRINT #<Dateinr.>, [USING <Maske\$>] <Text\$> [;|,]**
    - Schreiben eines Text-Strings in eine Datei {6/294}; Syntax des PRINT-Befehls ist identisch mit der im Kapitel 'Textanzeige, Farben' unter PRINT geschilderten Syntax für Bildschirmausgaben.
    - Die Daten werden an der aktuellen Dateizeigerposition ohne Anführungszeichen in die Datei eingefügt; dahinter wird <CR+LF> eingetragen. Der Dateizeiger wird anschließend inkrementiert, d.h. auf die nächste Datensatz-Einfügestelle gesetzt.
    - Die Datei muss vorher einmal in der Zugriffsart OUTPUT oder APPEND geöffnet worden sein.
    - Die Syntax für USING <Maske\$> ist dieselbe wie beim Befehl PRINT USING (siehe Kapitel 'Textanzeige, Farben').
  - **LINE INPUT #<Dateinr.>, <Stringvariable\$>**
    - Lesen eines Datensatzes (bis zum nächsten Enter= CR+LF) aus der Datei in die Stringvariable. Im Gegensatz zu INPUT werden Kommas ebenfalls eingelesen und nicht als Trennzeichen interpretiert. Der Datensatz wurde ursprünglich typischerweise mit PRINT abgespeichert.
    - Die Daten werden an der aktuellen Dateizeigerposition aus der Datei gelesen. Der Dateizeiger wird anschließend inkrementiert, d.h. auf die nächste Datensatz-Lesestelle gesetzt.
    - Die Datei muss vorher einmal in der Zugriffsart INPUT geöffnet worden

- sein.
- `<StringvariableS> = INPUTS (<AnzahlZeichen%>), <Dateinr. ohne #>`
    - Lesen einer wählbaren Anzahl von Zeichen (inklusive Kommas und CR+LF) aus der Datei in eine Stringvariable {9/114} {11/306}, anschließend den Dateizeiger inkrementieren
  - EOF (<Dateinr. ohne #>) - Funktion, liefert True (-1) zurück, nachdem der letzte Datensatz gelesen wurde (EOF = "End Of File").
  - LOF (<Dateinr. ohne #>) - Funktion, liefert die Anzahl der in der Datei gespeicherten Bytes zurück (max 2^31-1; LOF = "Length Of File").
  - Befehle zum Bearbeiten des Dateizeigers (bei sequentiellen Dateien nicht besonders hilfreich, da der Dateizeiger Byte- und nicht Datensatz-orientiert ist 11/465)):
    - LOC (<Dateinr>) - liefert die aktuelle Byte-Position des zuletzt gelesenen oder geschriebenen Datensatzes geteilt durch 128
    - SEEK (<Dateinr>) - liefert die Byte-Position des nächsten zu lesenden oder zu schreibenden Datensatzes zurück (1. Byte in der Datei hat die Nummer '1')
    - SEEK (<Dateinr>, <PositionS>) - setzt den Dateizeiger für den nächsten zu lesenden oder zu schreibenden Datensatzes auf die angegebene Byte-Position.
  - Beispiel: Schreiben und Lesen von 2 Datensätzen, die in je 2 Felder unterteilt sind (ein Textfeld für den Namen und ein numerisches Feld für den Geburtstag; siehe auch FILE-SEQ.BAS):
 

```

OPEN "birth.dat" FOR OUTPUT AS #1 'Datei zum Schreiben öffnen
WRITE #1, "Thomas", 28.01 'existiert die Datei birth.dat
WRITE #1, "Marlies", 29.02 'schon, so wird der Inhalt gelöscht!
CLOSE #1
OPEN "birth.dat" FOR INPUT AS #1 'Datei zum Lesen öffnen
FOR i = 1 TO 2
  INPUT #1, name$(i), birthday!(i)
  PRINT name$(i), birthday!(i)
NEXT i
CLOSE #1 'Datei schließen
KILL "birth.dat" 'Datei löschen
```
  - Wie kann ich einen Datensatz inmitten einer Sequentiellen Datei lesen, schreiben oder einfügen? Dies ist eine der am meisten gestellten Fragen in allen QBasic-Foren. Im Gegensatz zu einer Direktzugriffs-Datei ist dies nicht mit einem einzigen Befehl möglich. Sie müssen zunächst alle vorhergehenden Datensätze in einen Puffer oder eine Pufferdatei einlesen. Dann hängen Sie den geänderten oder einzufügenden Datensatz hinten an den Pufferinhalt an. Schließlich lesen Sie die restlichen Datensätze aus der Sequentiellen Datei in den Puffer ein. Zum Schluss überschreiben Sie die Sequentielle Datei mit dem Pufferinhalt. Bei Verwendung einer Pufferdatei können Sie diese nach Löschen der Sequentiellen Datei mit deren Namen umbenennen.
- \*\*\*\*\*  
 \* Direktzugriffs-Dateien mit TYPE-Puffer {9/118+123}  
 \*\*\*\*\*
- Allg. Hinweise:
    - Siehe auch Kapitel 'Dateibearbeitung - Allgemeines..' und FILE-TYP.BAS.
    - Eine Direktzugriffs-Datei besteht aus Datensätzen fester Länge, die in Felder ebenfalls fester Länge unterteilt sein können. Die Felder können sich im Datentyp und in der Länge voneinander unterscheiden. Die Datensätze lassen sich über ihre jeweilige Datensatznummer (ab 1) ansprechen.
    - Als Zwischenpuffer (vorübergehender Speicher) für die aus der Datei gelesenen und in die Datei geschriebenen Datensätze dient ein anwenderdefiniertes Feld, das mit TYPE...END TYPE deklariert werden kann (siehe unten).
    - Strings, die kürzer sind als deklariert, werden beim Schreiben automatisch rechts mit Leerzeichen (Blanks) aufgefüllt, die nach dem Lesen mit RTRIMS wieder beseitigt werden können.
  - TYPE <Name des Typs> <Elementname1> AS <Typ> [<Elementname2> AS <Typ>] ... END TYPE
    - Anwenderdefinierten Datentyp deklarieren (mehrdimensionales Feld gemischten Datentyps; s. Kapitel 'Felder'; muss im Hauptprogramm stehen).
  - DIM <FeldnameS> (<Feldlänge%> AS <Name des Typs>)
    - gemischtes Feld deklarieren; kann auch in einer SUB/ FUNCTION stehen; siehe Kapitel 'Felder'.
  - OPEN [PfadnameS] <DateinameS> FOR RANDOM AS #<Dateinr. 1...255> LEN = <Anzahl Bytes je Datensatz> - Direktzugriffs-Datei öffnen

- PUT #<Dateinr.>, <Datensatznr&>, <FeldnameS> (<Nr% des Feldelements%>)
    - Datensatz mit der <Datensatznr.> aus RAM-Feld (anwenderdefiniertes gemischtes Feld) in die Datei schreiben. Der erste Datensatz hat die Nummer 1, nicht 0.
  - CLOSE [#<Dateinr.>]
    - Datei schließen; muss bei jedem Wechsel zwischen PUT und GET erfolgen. Bei weggelassener Dateinr. werden alle offenen Dateien geschlossen.
  - GET #<Dateinr.>, <Datensatznr&>, <FeldnameS> (<Nr% des Feldelements%>)
    - Datensatz mit der <Datensatznr.> aus der Datei ins RAM-Feld (anwenderdefiniertes gemischtes Feld) einlesen.
  - LOF (<Dateinr. ohne #>) - Funktion, liefert die Anzahl der in der Datei gespeicherten Bytes zurück (max 2^31-1; LOF = "Length Of File").
  - EOF - End-Of-File-Funktion funktioniert bei Direktzugriffs-Dateien nicht!
  - Befehle zum Bearbeiten des Dateizeigers {11/465}:
    - LOC (<Dateinr.>) - liefert die Nummer des zuletzt gelesenen oder geschriebenen Datensatzes zurück
    - SEEK (<Dateinr.>) - liefert den aktuellen Inhalt des Dateizeigers zurück, d.h. die Nummer des nächsten zu lesenden bzw. zu schreibenden Datensatzes
    - SEEK <Dateinr>, <DatensatznrS> - setzt den Dateizeiger für den nächsten Schreib-/ Lesevorgang auf die angegebene Datensatznummer
  - Beispiel: Einen gemischten Datensatz in die Datei "meinquiz.dat" schreiben und wieder rücklesen (siehe auch FILE-TYP.BAS):
 

```

TYPE quiz 'Datentyp "quiz" deklarieren: Feld m je
frage AS STRING * 70 '3 String-Elementen (70, 50 und 50 Zeichen lang)
antw1 AS STRING * 50 'und einem Integer-Element,
antw2 AS STRING * 50 '(2 Bytes) ==> in Summe 172 Bytes
oknr AS INTEGER 'Deklaration muss im Hauptprogramm stehen!
END TYPE
DIM geschichte (1 TO 20) AS quiz 'Geschichtsqiz-Feld v. Typ "quiz" mit 20
' Feldelemente deklarieren (auch in SUB oder
' FUNCTION möglich)
OPEN "meinquiz.dat" FOR RANDOM AS #1 LEN = 172 'Datei öffnen
PUT #1, 13, geschichte(13) '13. Element aus dem Feld geschichte in den
' 13. Datensatz der Datei meinquiz.dat
' transferieren
CLOSE #1 'Datei schließen und ...
OPEN "meinquiz.dat" FOR RANDOM AS #1 LEN = 172 '... erneut öffnen
GET #1, 13, geschichte(13) '13. Element aus der Datei ins Feld geschichte
' einlesen
CLOSE #1 'Datei schließen
```
- \*\*\*\*\*  
 \* Direktzugriffs-Dateien mit FIELD-Puffer {9/118+139}{11/343}  
 \*\*\*\*\*
- Allgemeine Hinweise:
    - Siehe auch Kapitel 'Dateibearbeitung - Allgemeines..', FILE-FLD.BAS sowie {5/65}.
    - Vor dem Schreiben in die Datei muss ein Datensatz mit dem speziellen LSET-Befehl in den FIELD-Puffer eingetragen werden (siehe unten).
    - Ein Datensatz besteht aus einem oder mehreren Field-Elementen (Datensatzfeldern)
    - Die enorme Flexibilität dieser Dateiart liegt darin, dass die Längen und Namen der Feldelemente im FIELD-Puffer noch zur Laufzeit beliebig manipuliert werden können, so dass ein Anlegen und Bearbeiten beliebiger Datenbankstrukturen möglich ist, die zum Zeitpunkt der Programmentwicklung noch gar nicht bekannt sein müssen. Man kann sich das so vorstellen, dass das Programm auf Tabellen zugreifen kann, deren Zeilen unterschiedlich viele Spalten haben, welche sich wiederum von der Breite her unterscheiden können; siehe {9/139}. Dies ändert jedoch nichts an der Tatsache, dass eine einmal geöffnete konkrete Direktzugriffs-Datenbankdatei grundsätzlich nur gleich lange Datensätze speichern kann!
  - OPEN [PfadnameS] <DateinameS> FOR RANDOM AS #<Dateinr. 1...255> LEN = <Anzahl Bytes je Datensatz>
    - Direktzugriffs-Datei öffnen
  - FIELD #<Dateinr.>, <Field-Elementlänge1%> AS <Field-Elementname1S> [<Field-Elementlänge2%> AS <Field-Element-Name2S>]...
    - FIELD-Puffer für einen Datensatz deklarieren, u.U. bestehend aus mehreren Field-Elementen (Längen in Bytes). Ein Field-Element kann nur Strings

enthalten. Numerische Werte müssen vor ihrem Eintrag in den FIELD-Puffer mit einem MKxS-Befehl (s.u.) in "Pseudostrings" umgewandelt werden. Das Rückwandeln in numerische Werte nach dem Lesen erfolgt über einen entsprechenden CVx-Befehl (s.u.).

- {MKIS|MKLS|MKXS|MKDS} (numerische Variable)
  - Aus einer INTEGER|LONG|SINGLE|DOUBLE-Variablen Pseudostrings gleicher Länge erzeugen, die in einen FIELD-Puffer eingetragen werden können ("Intel-Format": Low- vor High-Byte)
- LSET <Field-Elementname\$> = <Variable\$>
  - In ein Field-Element innerhalb eines FIELD-Puffers eine Variable eintragen (Wertzuweisung). Überschüssige Zeichen werden rechts abgeschnitten, kurze Strings rechts mit Leerzeichen (Blanks) aufgefüllt.
- RSET <Field-Elementname\$> = <Variable\$>
  - wie LSET, jedoch rechtsbündige Anordnung; Überschüssige Zeichen werden links abgeschnitten; weniger gebräuchliche Variante.
- PUT #<Dateinr.>, <Datensatznr.>
  - Datensatz aus dem FIELD-Puffer in den Datensatz mit der <Datensatznr.> in die Datei schreiben.
- CLOSE [#<Dateinr.>]
  - Datei schließen, muss bei jedem Wechsel zwischen PUT und GET erfolgen. Bei weggelassener Dateinr. werden alle offenen Dateien geschlossen.
- GET #<Dateinr.>, <Datensatznr.>
  - Inhalt des Datensatz mit der <Datensatznr.> aus der Datei in den FIELD-Puffer transferieren.
- {CVI|CVL|CVS|CVD} (<String 2...8 Bytes>)
  - Pseudostring aus einem gelesenen FIELD-Puffer wieder in numerische Werte rückwandeln.
- <Variable> = <Field-Elementname\$>
  - gelesenes Datensatz-Field-Element aus dem FIELD-Puffer lesen und in eine Variable eintragen. Beispiele:
    - anna\$ = feld1\$ 'Stringvariable
    - otto% = CVI(feld2\$) 'numer. Variable, muss vorher rückgewandelt werden
- LOF (<Dateinr., ohne #>)
  - Funktion, liefert die Anzahl der in der Datei gespeicherten Bytes zurück (max 2^31-1; LOF = "Length Of File").
- EOF - End-Of-File-Funktion funktioniert bei Direktzugriffs-Dateien nicht!
- Befehle zum Bearbeiten des Dateizeigers {11/465}:
  - LOC (<Dateinr.>) - liefert die Nummer des zuletzt gelesenen oder geschriebenen Datensatzes zurück
  - SEEK (<Dateinr.>) - liefert den aktuellen Inhalt des Dateizeigers zurück, d.h. die Nummer des nächsten zu lesenden bzw. zu schreibenden Datensatzes
  - SEEK (<Dateinr.>, <Datensatznr\$>) - setzt den Dateizeiger für den nächsten Schreib-/ Lesevorgang auf die angegebene Datensatznummer.
- Beispiel: (siehe FILE-FLD.BAS und {11/343}):
 

'Bearbeitung einer Telefon-Datenbank: Ein Name (Textstring) und eine Telefonnummer werden in einen FIELD-Puffer eingetragen und dann von dort in den dritten Datensatz der Datenbank-Datei "Telefon" transferiert:

```

OPEN "telefon" FOR RANDOM AS #1 LEN = 20 'Telefon-Datenbank, Länge 12+8=20
FIELD #1, 12 AS name$, 8 AS no$ 'FIELD-Puffer für 1 Datensatz deklarieren
' mit 12 Bytes für Namen und 8 Bytes für
' Telefonnummer
LSET name$ = "antoni" 'Namen in FIELD-Puffer eintragen
LSET no$ = MKL$(23852) 'Telefonnummer (LONG Integer) in
' String wandeln u. in FIELD eintragen
PUT #1, 3 'Inhalt des FIELD-Puffers in den 3. Daten-
' satz der Datei schreiben
CLOSE #1
OPEN "telefon" FOR RANDOM AS #1 LEN = 20 'wie oben, Datei öffnen zum Lesen
FIELD #1, 12 AS name$, 8 AS no$ 'wie oben; muss nochmals deklariert werden!
GET #1, 3 '3. Datensatz in den FIELD-Puffer lesen
PRINT name$, "Telefon-Nr. "; CVL(no$) 'Inhalt des FIELD-Puffers anzeigen;
' numerischen 'Pseudostring' no$ vorher rückwandeln

```

```

*****
* Binäre Dateien {5/65} {11/301}
*****
- Allg. Hinweise:
  - Siehe auch Kapitel 'Dateibearbeitung - Allgemeines.', FILE-BIN.BAS
  und FILECOPY.BAS
  - Binäre Dateien sind quasi Byte-Felder ohne besondere Datensatzstruktur.
  Die Datenzugriffe erfolgen über einen Dateizeiger an beliebiger Position
  oder fortlaufend. Das erste Byte hat die Position 1, das letzte Byte die
  Position LOF(<Dateinr.>)
- OPEN [Pfadname$] <Dateiname$> FOR BINARY AS #<Dateinr. 1...255> -
  Binäre Datei zum Lesen und/oder Schreiben öffnen und Dateizeiger auf 1 set-
  zen (d.h. aufs erste Byte in der Datei). Beim Wechsel zwischen Lesen und
  Schreiben (PUT und GET) muss die Datei nicht geschlossen werden.
- PUT #<Dateinr.>, [<Position>], <Variable> -
  Variable ab der aktuellen Position des Dateizeigers [bzw. ab der
  angegebenen Position] in die Datei hineinschreiben und anschließend den
  Dateizeiger hinter das letzte geschriebene Byte setzen.
  Textvariablen müssen vorher über DIM text AS STRING * <Länge> mit der
  richtigen festen Länge deklariert werden
- GET #<Dateinr.>, [<Position>], <Variable> -
  Daten ab der aktuellen Position des Dateizeigers [bzw. ab der ange-
  gebenen Position] in eine Variable lesen und anschließend den Dateizeiger
  hinter das letzte gelesene Byte setzen.
  Textvariablen müssen vorher über DIM text AS STRING * <Länge> mit der
  richtigen festen Länge deklariert werden
- Befehle zum Bearbeiten des Dateizeigers {11/465}:
  - LOC (<Dateinr.>) - liefert die Position ('Location') des zuletzt gelesenen
  oder geschriebenen Bytes zurück (max. 2^31 - 1; das erste Byte der Datei
  hat die Nummer '1')
  - SEEK (<Dateinr.>) - liefert die Byte-Position des nächsten zu lesenden
  bzw. zu schreibenden Bytes zurück
  - SEEK <Dateinr.>, <Byteposition&>
    - Dateizeiger auf eine wählbare Byteposition setzen
- CLOSE [#<Dateinr.>] - Datei schließen
  Bei weggelassener Dateinr. werden alle offenen Dateien geschlossen.
- EOF (Dateinr. ohne #) - Funktion, liefert True (-1) zurück, nachdem das
  letzte Byte gelesen wurde (EOF = "End Of File").
- LOF (<Dateinr. ohne #) - Funktion, liefert die Anzahl der in der Datei ge-
  speicherten Bytes zurück (max 2^31-1; LOF = "Length Of File").
- Beispiel 1: Hex-Zahl 4711h (=18193) ab Byte 33 in Datei yyy.bin hinterlegen
  und wieder auslesen (siehe auch FILE-BIN.BAS):
  OPEN "yyy.bin" FOR BINARY AS #3
  z& = &H4711
  SEEK #3, 33 'Dateizeiger auf das 33. Byte setzen
  PUT #3, , z& 'LONG-Integerzahl in Byte 33..36 der Datei schreiben
  GET #3, 33, y&: PRINT y&
  PRINT "Die Datei yyy.bin ist"; LOF(3); " Bytes lang"
  CLOSE #3
- Wollen Sie einzelne Bytes lesen oder schreiben, so stoßen Sie auf das
  Problem, dass QBasic keinen Datentyp "BYTE" kennt. Sie können sich behelfen
  indem Sie die Bytes als Strings der Länge 1 behandeln und mit
  DIM <bytevariable> AS STRING * 1
  dimensionieren. Mit ASC() und CHR() kann man Strings in Zahlen umwandeln
  und umgekehrt.
- Beispiel 2: Kopieren einer Datei "Datei1" in die Datei "Datei2";
  siehe auch FILECOPY.BAS
  DIM t AS STRING * 1 'Byte-Variablen t anlegen ("Pseudo-String")
  OPEN "Datei1" FOR BINARY AS #1
  OPEN "Datei2" FOR BINARY AS #2
  DO UNTIL LOC(1) = LOF(1) 'Schleife ueber alles Bytes
  GET #1, , t 'Byte lesen; beachte die beiden Kommas!
  PUT #2, , t 'Byte schreiben
  LOOP
  CLOSE

```

\*\*\*\*\*  
 \* Druckerausgabe  
 \*\*\*\*\*  
 Die Druckerausgabe funktioniert von QBasic aus nur mit Druckern, die am Parallelport hängen und den ASCII-Zeichensatz mit deutschen Umlauten verstehen, z. B. bei Druckern mit Epson FX80- oder IBM Proprinter-Emulation.  
 - LPRINT <Text\$> [;|. ] - Text auf Drucker ausgeben {11/287} in neuer Zeile bzw. [direkt hinter dem letzten gedruckten Zeichen | am Beginn des nächsten 14-Spalten-Bereichs]. Die Syntax entspricht dem PRINT-Befehl für Bildschirmausgaben. Ein Öffnen des Druckers über OPEN ist beim LPRINT-Befehl nicht erforderlich.  
 - LPRINT USING <Maske\$>; <Ausdruck> [; <Ausdruck2> ...] - Formatierte Ausdrücke erzeugen (Tabellen usw.). Die Syntax ist dieselbe wie beim Befehl PRINT USING (siehe Kapitel 'Textanzeige, Farben').  
 - OPEN "LPT<DruckerNr%>:" FOR OUTPUT AS #<Dateinr> - Drucker für Ausgabe öffnen; die Ausgabe des Textes erfolgt mit WRITE oder PRINT, das Schließen des Druckers mit CLOSE - wie bei einer sequentiellen Datei; siehe auch Kapitel 'Sequentielle Dateien' sowie {11/375}.  
 - WIDTH LPRINT <Spaltenzahl> - legt die Länge der Ausgabezeilen fest {11/464}  
 - WIDTH "LPT<DruckerNr%>:"; <Spaltenzahl> - dito; z. B. WIDTH "LPT1:"; 72  
 - LPOS (<DruckerNr%>) - liefert die Anzahl der Zeichen zurück, die nach dem letzten <CR> (=CHR\$(13)) ausgegeben wurden {11/466}.  
 Windows-Drucker werden in der DOS-Box nicht unterstützt. Es gibt jedoch Tools, mit deren Hilfe sich beliebige Windows-Drucker - auch über den USB-Port - von DOS und QBasic aus ansprechen lassen. Das wohl beste Tool für Programmierer ist "RGH-Druck", das auf meiner Seite www.qbasic.de unter "Download | Tools" zusammen mit einem QBasic-Anwendungsbeispiel zum Herunterladen bereitsteht. Wenn Sie über den Druckerport externe Hardware wie LEDs, Relais usw. ansteuern wollen, so erhalten Sie alle wichtigen Informationen auf der Webseite www.FrankSteinberg.de.

\*\*\*\*\*  
 \* Serielle Schnittstellen {11/375}  
 \*\*\*\*\*  
 - OPEN "COM<Nr%>:" <Option1> <, Option2> ... AS #<Dateinr>  
 oder  
 OPEN "COM<Nr%>:" <Optionen> FOR <Modus> AS #<Dateinr> LEN =<Länge> - Serielle Schnittstelle als Datei <Dateinr> öffnen mit den folgenden Optionen (siehe auch Kapitel 'Dateibearbeitung...'):  
 - 75|110|150|300|600|1200|2400|4800|9600|19200 - Bitrate in Bits/s (Baud) 19200 Bits/s ist auch machbar, obwohl in der QBasic-Hilfe nicht dokumentiert.  
 - ,{N|E|O} - kein|gerades|ungerades Paritätsbit (None|Equal|Odd Parity)  
 - ,{4|5|6|7|8} - Anzahl der Datenbits ( Paritätsbit nicht mitgezählt; Vorbesetzung=7)  
 - ,{1|1.5|2} - Anzahl der Stop-Bits (Vorbesetzung=1)  
 - ,{ASC|BIN} - Öffnen für ASCII- | binäre Datenübertragung  
 - ,CD <AnzMillisec> - Wartezeit in Millisec für Steuersignal DCD (Data Carrier Detect) zur Erkennung der Verbindungsaufnahme.  
 - ,CS <AnzMillisec> - Wartezeit in Millisec für Steuersignal CTS (Clear To Send) zum Signalisieren der Sendebereitschaft.  
 - ,OP <AnzMillisec> - Wartezeit in Millisec für 'Open Com', bis die Verbindung hergestellt ist.  
 - ,LF - zusätzlicher Zeilenvorschub <LF> (=Linefeed = CHR\$(10)) nach Wagenrücklauf <CR> (= Carriage Return = CHR\$(13)) senden.  
 - ,{RB|TB} <AnzBytes%> - Größe des Empfangs- | Sendepuffers in Bytes festlegen (typisch z. B. 2048 Bytes)  
 - ,RS - Signal von der RTS-Leitung (Request to Send) unterdrücken (dient zur Sendeanfrage)  
 Beispiel für 'normale Konfiguration':  
 OPEN "COM2: 300, N, 8, 1, CDO, CS0, DSO, OPO, RS, TB2048, RB2048" FOR RANDOM AS #1  
 - Serial Port 2 öffnen mit 300 Baud, ohne Parity-Bit, mit 8 Datenbits und einem Stop-Bit, ohne Wartezeiten und Handshake, je 2048 Bytes für Send- und Empfangspuffer.  
 Hinweis: Das Kommunikationsprogramm muss zuerst auf dem Empfänger-PC, dann auf dem Sender-PC gestartet werden!  
 - LOC <Dateinr%> - Funktion, die die Nummer des Datensatzes zurückliefert, der gerade gesendet oder empfangen wird; bei binären Dateien die Nummer des aktuellen Bytes. Ist noch nichts empfangen, so liefert LOC eine '0' zurück.

- ON COM (<Nr%> GOSUB <Marke> - Ereignisgesteuertes Ansprechen einer lokalen Subroutine, wenn ein neues Zeichen empfangen wurde.  
 - COM {ON|OFF|STOP} - Ereignisverfolgung für serielle Schnittstelle aktivieren | deaktivieren | unterbrechen mit Speicherung  
 - WIDTH COM <Nr%>; <Spaltenzahl%> - legt die Länge von Text-Ausgabezeilen fest {11/464}  
 - LOF - liefert die Anzahl der freien Bytes im Sendepuffer zurück  
 PowerBASIC unterstützt auch die Ports COM3 und COM4 sowie höhere Bauraten bis zu 115000 Bits/s.  
 Wenn Sie über die serielle Schnittstelle externe Hardware wie LEDs, Relais, Schrittmotoren usw. ansteuern wollen, so erhalten Sie alle wichtigen Informationen auf den Webseiten www.FrankSteinberg.de und www.skilltronics.de.

\*\*\*\*\*  
 \* Direkter Speicherzugriff und I/O-Port-Zugriff {11/392}  
 \*\*\*\*\*  
 Speichermodell der 8x86-Prozessoren (Segment- und Offsetadressen)

Die 8x86-Prozessoren kennen im unteren ('konventionellen') 1 MB- Speicherbereich leider keine lineare Adressierung, sondern der Adressraum ist in 64 KB große Segmente aufgeteilt, zwischen denen über die Segmentadresse umgeschaltet werden muss. Die Bytes innerhalb eines Segments werden durch die Offsetadresse angesprochen. Die physikalische, auf dem externen Adressbus erscheinende Speicheradresse wird auf dem CPU-Chip hardwaremäßig aus der aktuellen Segment- und Offsetadresse gemäß der folgenden Gleichung gebildet:  
 Physikalische Adresse = Segmentadresse \* 16 + Offsetadresse  
 (0...2^20) (0...2^16) (0...2^16)

Ablage von QBasic-Variablen im Speicher

-----  
 - Numerische Variablen: werden direkt an der durch VARSEG und VARPTR abfragbaren Adresse abgelegt und zwar im 'Intel-Format': Low-Byte vor High-Byte und Low-Word vor High-Word. Siehe auch PEEKPOK1.BAS, PEEKPOK2.BAS und BIOSDAT.BAS.  
 Vorzeichenbehaftete Größen müssen vor und nach dem Speichern trickreich in Bytes umgewandelt werden (siehe untenstehendes Beispiel 1).  
 - Felder: Auf alle Felder (statische, dynamische und anwenderdefinierte) greift QBasic intern mittels spezieller "Feld-Deskriptoren" zu, die über PEEK und POKE nicht zugänglich sind.  
 - Statische Strings: Statische Strings sind Strings fester Länge, die mit ... AS STRING \* <Länge> deklariert sind. QBasic legt statische Strings direkt an der durch VARSEG(string\$) und VARPTR(string\$) abfragbaren Speicherposition ab. Statische Strings haben keinen String-Deskriptor {11/404}.  
 - Dynamische Strings: Alle implizit, d.h. ohne '... AS STRING \* <Länge>' deklarierten Strings, sind dynamische Strings. D.h. ihre Länge kann sich zur Laufzeit ändern. Der Zugriff auf dynamische Strings ist nicht direkt, sondern nur auf dem Umweg über einen String-Deskriptor möglich. Die Speicheradresse des String-Deskriptors für text\$ ist über VARSEG(text\$) und VARPTR(text\$) abfragbar (siehe {11/393}, {9/30} und das untenstehende Beispiel 2).  
 Der String-Deskriptor besteht aus zwei INTEGER-Werten: Die ersten beiden Bytes enthalten die Länge, die letzten beiden Bytes die Offset-Adresse des Strings. Der String befindet sich (außer bei PowerBASIC und Quick Basic) grundsätzlich immer in demselben Segment wie der Stringdeskriptor.  
 Hinweis zu PowerBASIC und QuickBASIC: Dort lassen sich die Adressen beliebiger Strings über SADD bzw. STRSEG/STRPTR direkt abfragen.

Bestimmung der absoluten Adresse von Variablen mit VARSEG und VARPTR

-----  
 - VARSEG (<Variablenname>) - Segmentadresse einer Variablen ermitteln (Wertebereich 0-65535; u. U. in LONG-Größe einlesen, da vorzeichenlos)  
 - VARPTR (<Variablenname>) - Offsetadresse einer numerischen Variablen oder einer statischen Stringvariablen ermitteln (Wertebereich 0-65535) bzw. Offsetadresse des String-Deskriptors einer dynamischen Stringvariablen (siehe oben).  
 - VARPTRS (<Befehlsstring\$>) - Selten verwendete Funktion zur Ermittlung der Stringadresse eines Befehlsstrings für den PLAY oder DRAW Befehl. Der Befehlsstring kann somit über einen Pointer mit vorangehendem "X" übergeben

- werden (Beispiel: PLAY "X" + VARPTR(<Variable\$>); siehe QBasic-Onlinehilfe.
- **VARPTRS (<Variablenname\$>)** - Selten verwendete Funktion zur Ermittlung des Typs und der Offsetadresse einer Variablen als String mit drei Zeichen (siehe {11/469} und VARPTR.BAS):
  - 1. Zeichen = Typ der Variablen: CHR\$(2|3|4|8|20) = INT|STRING|SINGLE|DOUBLE|LONG
  - 2. und 3. Zeichen = Offsetadresse der Variablen (bzw. des String-Deskriptors bei dynamischen Strings) als String: 2. Zeichen = CHR\$(Lo-Address), 3. Zeichen = CHR\$(Hi-Address)

#### Speicherbytes lesen und schreiben mit PEEK und POKE

Siehe auch PEEKPOK1.BAS, PEEKPOK2.BAS und BIOSDAT.BAS.

- Hinweis: Der direkte Speicherzugriff ist bei QBasic grundsätzlich nur Byte-weise möglich. Bei PowerBASIC wird über PEEKI und PEEKL auch ein Zugriff auf INTEGER- und LONG-Größen unterstützt.
- **DEF SEG = <Segmentadresse&>** Festlegen der aktuellen Segmentadresse (0...65536) für die folgenden PEEK und POKE-Befehle zum Schreiben/ Lesen von Speicherbytes
- **DEF SEG** - Wenn die Segmentadresse weggelassen wird, setzt DEF SEG die Segmentadresse wieder auf das QBasic-Standard-Datensegment zurück.
- **PEEK (<Offsetadresse&)** - Lesen eines Speicherbytes: Funktion vom Typ INTEGER, die im Low-Byte den Inhalt des durch die angegebene Offsetadresse adressierten Speicherbytes im aktuellen Segment zurückliefert. Die aktuelle Segmentadresse lässt sich durch DEF SEG verändern (siehe oben).
- **POKE <Offsetadresse&,> <Wert%>** - Schreiben eines Speicherbytes: Das niederwertige Byte von Wert% wird in das durch die Offsetadresse& adressierte Speicherbyte im aktuellen Segment geschrieben. Die aktuelle Segmentadresse lässt sich durch DEF SEG verändern (siehe oben).

#### Beispiele für die obengenannten Befehle zum direkten Speicherzugriff

- Beispiel 1: Variable e% mit PEEK lesen, inkrementieren und mit POKE zurück-schreiben (siehe auch PEEKPOK1.BAS):
 

```
a%=-4711 'bezüglich CVI und MKIS: Siehe Kapitel 'Dateien mit FIELD-Puffer'
segm& = VARSEG(a%) 'Segmentadresse von a%
offs& = VARPTR(a%) 'Offsetadresse von a%
DEF SEG = segm& 'aktuelles Segment:=Segment, in d.sich a% befindet
b$ = CHR$(PEEK(offs&)) + CHR$(PEEK(offs& + 1))
      'Lo-/Hi-Byte als 'Pseudostring' lesen (Trick!)
c% = CVI(b$) + 1 'Pseudostring wieder in INTEGER-Wert wandeln u.inkrem.
'---- Inkrementierten Wert in a% zurückspeichern per POKE und anzeigen -----
d$ = MKIS(c%) 'inkrementierten Wert in Pseudostring umandeln; Trick!
POKE offs&, c% 'POKE speichert immer nur das Lo-Byte
hibyte% = CVI(RIGHT$(d$, 1) + CHR$(0)) 'Hi-Byte ins Lo-Byte schieben
POKE offs& + 1, hibyte% 'Hi-Byte speichern
PRINT a% 'Angezeigt wird der inkrementierte Wert -4710
DEF SEG 'Standard-Datensegment wieder aktivieren
```

- Beispiel 2: Dynamischen String text\$ über Deskriptor lesen, ändern u. zurück-schreiben (siehe auch PEEKPOK2.BAS):
 

```
text$ = "A-Hörnchen" 'Textstring abspeichern
segm& = VARSEG(text$) 'Segmentadresse des Deskriptors von text$ ermitteln
offs& = VARPTR(text$) 'Offsetadresse des Deskriptors von text$ ermitteln
DEF SEG = segm& 'aktuelles Segment := Segment, in dem sich sowohl der
      'String als auch der String-Deskriptor befindet
stringadr& = CLNG(PEEK(offs& + 3)) * 256 + PEEK(offs& + 2)
      'Hi- und Lo-Byte der eigentlichen Stringadresse aus
      'Byte 3 und 4 des String-Deskriptors lesen; CLNG kon-
      'vertiert INTEGER zu LONG (vermeidet Überlauf)
ersteszeichen% = PEEK(stringadr&) + 1 '1. Zeichen d.Strings lesen u.
      'inkrementieren (aus "A" wird "B")
POKE stringadr&, ersteszeichen% 'geändertes 1. Zeichen zurückspeichern
DEF SEG 'Standard-Datensegment wieder aktivieren
```

#### Speicherbereich mit BSAVE/BLOAD in Datei schreiben und aus Datei lesen {11/401}

- **BSAVE <Dateiname\$>, <Offsetadresse&>, <AnzahlBytes%>** - Ab der angegebenen Offsetadresse eine wählbare Anzahl von Speicherbytes in eine Datei schreiben. Die Datei braucht nicht explizit geöffnet und geschlossen zu werden. Das aktuelle Segment ist über DEF SEG anwählbar (siehe oben unter

- ' Speicherbytes lesen und schreiben mit PEEK und POKE').
- **BLOAD <Dateiname\$> [, <Offsetadresse&>]** - Mit BSAVE gesicherte Speicherbytes aus der Datei lesen und wieder an der alten Stelle [bzw. an der angegebenen Offsetadresse] im Speicher ablegen. Die Datei braucht nicht explizit geöffnet und geschlossen zu werden.
- Beispiel für BSAVE/BLOAD: Inhalt des Farb-Textbildschirms SCREEN 0 in die Datei xxx.bld speichern und anschließend wieder restaurieren (siehe 11/400 und BSAVE1.BAS):
 

```
DEF SEG = &HB800 'Segmentadresse des Farbbildschirms= B800 hex
LOCATE 12, 30: PRINT "Dies wird gerettet": SLEEP
BSAVE "xxx.bld", 0, 4000 'Bildschirminhalt 4KBytes sichern nach xxx.bld
CLS : PRINT "Nix mehr da!!": SLEEP
BLOAD "xxx.bld": SLEEP 'gesicherten Bildschirminhalt wiederherstellen
DEF SEG 'Standard-Datensegment wieder aktivieren
```

#### Externe Maschinenspracheprogramme aufrufen {11/402}

- **CALL ABSOLUTE <Offsetadresse&>** - Externes Maschinenspracheprogramm unter der angegebenen Offsetadresse aufrufen (Segmentadresse kann durch DEF SEG definiert werden; siehe oben).
- **CALL ABSOLUTE (<Parameter 1>, <Parameter 2>, ... <Offsetadresse&>)** - wie oben, jedoch mit Übergabe von Parametern.
- Hinweise zu QuickBASIC und PowerBASIC: Über **CALL INTERRUPT** können System-Interrupt-Routinen direkt angesprochen werden. Bei Verwendung von CALL ABSOLUTE muss QuickBASIC mit 'QB /L' aufgerufen werden, um die Quick-Library QB.QLB einzubinden.

#### Zugriff auf I/O-Ports {11/467}

- Die CPUs der x86-er-Familie unterstützen den Zugriff auf Hardware-Komponenten wie Timerbausteine und Tastatur über spezielle Hardware-Ein-/Ausgänge, die so genannten "I/O-Ports".
  - **INP (<I/O-Adresse%>)** - Byte von I/O-Port lesen (ähnlich PEEK)
  - **OUT <I/O-Adresse%>, <Wert%>** - niederwertiges Byte von Wert% zum I/O-Port senden (ähnlich POKE); Beispiel: OUT &H42, LSB% 'Speaker-Port ansteuern, d.h. I/O-Adresse 42 Hex
  - **WAIT <I/O-Adresse%>, <AND-Bitmuster%> [, <XOR-Bitmuster%>]** - Programm solange anhalten bis am I/O-Port die Bitkombination des AND-Bitmusters erscheint [bzw. die mit dem XOR-Bitmuster Exklusiv-oder-verknüpfte Bitkombination]
- Beispiele:
- Einlesen des Zeitwertes vom programmierbaren Intervall-Timer (PIT) 8253/8254 (dieser wird alle 0,838096515 Mikrosekunden inkrementiert, d.h. um 1 erhöht):
 

```
LoByte = INP(64): HiByte = INP(64): zeit = LoByte + HiByte * 256
```

 Bezüglich der Erzeugung kleiner Wartezeiten siehe auch www.FrankSteinberg.de.
  - Einlesen des Tastatur-Statusregisters: TastCode = INP(&H60)

#### Zugriff auf Gerätetreiber {11/468}

- **IOCTLS (#<Dateinr.>)** - Steuerzeichen (Statusdaten) von einem Gerätetreiber empfangen
- **IOCTL #<Dateinr.>, <Steuerzeichenfolge\$>** - Steuerzeichen an einen Gerätetreiber senden

#### Vorhandenen freien Speicherplatz für Variablen und Stack abfragen und ändern

- **FRE (0|-1|-2)** - vorhandenen Speicherplatz für Stringvariablen|numerische Variablen|Stack rückmelden. Insgesamt stehen ca. 30 KB Speicherplatz für Strings zur Verfügung {11/251+279+282}.
- **FREE("")** - bewirkt ein Aufräumen des String-Speichers ("Garbage Collection") und kann eventuell zusätzlichen Speicher für Stringvariablen freigeben.
- **CLEAR, <AnzahlBytes>** - Speicherplatz für den Stack in gewünschter Größe reservieren und initialisieren; Startwert für Stackgröße = 1200 Bytes.

\*\*\*\*\*

\* Umstieg von QBasic nach MS QuickBASIC V4.5 {11/482}

\*\*\*\*\*

- Vorteile von QuickBASIC gegenüber QBASIC:

- echter Compiler, erstellt ausführbare EXE-Dateien
- unterstützt Module und Bibliotheken, Quelltexte aus anderen Dateien über 'SINCLUDE' einbindbar
- einige zusätzliche Befehle (siehe unten)
- eine wesentlich ausführliche Online-Hilfe

- Portieren von QBasic-Programmen nach QuickBASIC:

QBasic-Programme sind problemlos auch unter QuickBASIC ablauffähig und zu EXE-Dateien kompilierbar; bei Verwendung des CALL ABSOLUTE Befehls wird jedoch die Quick-Library QB.QLB benötigt, und QuickBASIC muss über 'QB /L' aufgerufen werden (z.B. bei vielen Mausroutinen; ebenfalls erforderlich bei Verwendung von INTERRUPT[X] usw.) {9/6}

- Zusätzliche Befehle und Schlüsselwörter bei QuickBASIC:

- 'SINCLUDE'** - Compiler-Anweisung zum Einfügen von Quelltext aus einer anderen Datei (Include-Datei)
- ALIAS** - Verweist auf den Namen einer 'Nicht-BASIC-Prozedur'
- BYVAL** - Bewirkt 'Call by Value' statt 'Call by Reference' für einen Parameter, der an eine Nicht-Basic-Prozedur übergeben wird
- CDECL** - Bewirkt die Parameterübergabe an eine Prozedur gemäß C-Konventionen
- CALLS** - Aufruf von Subroutinen, die in anderen Programmiersprachen geschrieben wurden (Nicht-Basic-Prozeduren)
- COMMANDS** - Liefert die Befehlszeile zurück, mit der ein QuickBASIC-EXE-Programm aufgerufen wurde und ermöglicht so, Übergabeparameter abzufragen (siehe 'Parameterübergabe' im Kapitel 'Bedienung...')
- LOCAL|SIGNAL-** für künftige Anwendungen reservierte Schlüsselwörter
- SADD** - Offsetadresse einer Stringvariablen (Siehe Kapitel 'Direkter Speicherzugriff')
- INTERRUPT|INTERRUPTX** - direkter Systeminterrupt-Aufruf
- SETMEM** - Verändern des 'Far-Heap'-Speicherbereichs
- UEVENT|EVENT-** Anwenderdefinierte Ereignisverfolgung

\*\*\*\*\*

\* Umstieg von QBasic nach PowerBASIC V3.5 {11/485}

\*\*\*\*\*

- Vorteile von PowerBASIC gegenüber QBASIC:

- echter Compiler, erstellt ausführbare EXE-Dateien
- integrierter Inline-Assembler vorhanden
- unterstützt Module, Bibliotheken und Units
- mehr Datentypen (BCD, erweiterte Genauigkeit, siehe Kapitel 'Datentypen')
- nahezu beliebig große dynamische Strings, huge Arrays (Riesenfelder)
- EMS-Speicher-Support
- indirekte Adressierung über Pointer möglich
- Direktbearbeitung von Feldern (ARRAY SORT|SCAN...; siehe Kap.. 'Felder')
- TSRs erstellbar (speicherresidente Programme)
- höhere Geschwindigkeit (ca. 2\* schneller als QuickBASIC-EXE-Programme)
- wesentlich mehr Befehle, z.B. **MIN**, **MAX**, **ROUND**, **PEEKI** für Integer-Zugriff, **PEEKL** für Long-Integer-Zugriff, **PEEKs** und **POKES** für String-Zugriffe, Bit-Befehle und Befehle für die Bearbeitung kompletter Felder (sortieren, suchen, einfügen und löschen mit **ARRAY {SORT | SCAN | INSERT | DELETE}**)
- Die seriellen Ports lassen sich besser nutzen: Auch die Com-Ports 3 und 4 sowie Baudraten bis 115 kBits/s werden unterstützt.
- explizite Variablendeklarationen sind in der Entwicklungsumgebung erzwingbar über **<Options | Compiler | Variable declarations >** oder über den Metabefehl **SDIM ALL**.
- Unterstriche (Underscores) in Variablennamen und Sprungmarken sind zulässig (z.B. "Personen\_Konto")
- Ein PowerBASIC-Befehl darf sich über mehr als eine Zeile erstrecken. In einer fortzusetzenden Zeile muss man am Ende ein " \_" einfügen
- C-Bibliotheken lassen sich einbinden
- Verschiedene Compiler-Optimierungs-Optionen wählbar (nach Geschwindigkeit oder nach Programmgröße)

- Nachteile von PowerBASIC gegenüber QBASIC:

- PowerBASIC kostet ca. 99 US\$, während QBasic quasi Freeware ist. Bei [www.powerbasic.com](http://www.powerbasic.com) steht aber der etwas eingeschränkte Freeware-Compiler "FirstBASIC" zur Verfügung, der jedoch nicht mit der Maus bedienbar ist.
- PowerBASIC hat eine etwas weniger komfortable Entwicklungsumgebung. Die sofortige Syntax-Kontrolle beim Eintippen des Quelltextes fehlt. Profis werden das weniger vermissen als Einsteiger. Die Maus wird erst ab der PowerBASIC-Version V3.5 unterstützt.
- Es gibt keine eigenen Editierfenster für SUBs und FUNCTIONS. Diese müssen mühselig in der Quellsprachdatei aufgesucht werden.
- Die deutschsprachige Internet-Community ist bei QBasic wesentlich größer und lebendiger als bei PowerBASIC.
- Portieren von QBasic-Programmen nach PowerBASIC:
  - In DIM-Felddeklarationen 'TO' durch ':' ersetzen
  - 'DIM' und 'COMMON' vor SHARED-Anweisungen entfernen {11/279}
  - Nur INTEGER-Konstanten verwendbar. Bei diesen muss 'CONST' durch '%' ersetzt werden, z.B. **%anz=37** statt **CONST anz%=37**. Andere Konstanten-Typen gibt es nicht. Als Notbehelf kann man die Konstanten in DATA-Zeilen auslagern.
  - Anwenderdefinierte Verbundfelder (Typendeklarationen **TYPE ... END TYPE**) sind erst ab V3.5 möglich und müssen bei älteren PowerBASIC-Versionen entfernt werden. Statt dessen Einzeldeklarationen, FIELD-Puffer oder Flex-Strings verwenden (siehe Kapitel 'Felder').
  - Bei CASE-Anweisungen eventuell vorhandenes 'IS' entfernen.
  - 'EXIT DO' durch 'EXIT LOOP' ersetzen
  - Subroutinen-Aufrufe immer mit CALL und Parameterklammern versehen.
  - DECLARE-Anweisungen für SUBs und FUNCTIONS, die sich in derselben Datei befinden, im Hauptprogramm entfernen oder Parameterliste nur aus Typenbezeichnungen statt Namen zusammensetzen (z.B. **SINGLE** statt **egon!**).
  - SLEEP durch DELAY ersetzen bei PowerBASIC-Versionen < V3.5
  - SCREEN 13 wird nicht direkt, sondern nur mit Spezial-Routinen bzw. Bibliotheken unterstützt.
  - Sprungmarken müssen in einer extra Zeile stehen.
  - Vor Abfrage der Joystick-Feuerknöpfe mit STRIG muss die Ereignisverfolgung durch STRIG ON aktiviert werden.
  - Die Erkennung von Laufzeitfehlern erfolgt nicht automatisch, sondern muss am Programmfang gezielt mit **SERROR ALL ON** eingeschaltet werden. Oder man aktiviert in der Entwicklungsumgebung den Menüpunkt **<Options | Compiler | Error Test>**.

\*\*\*\*\*

\* Tipps zu häufig vorkommenden Programmierproblemen

\*\*\*\*\*

Unterstützung für nahezu alle nur denkbaren Programmierprobleme gibt es auf [www.qbasic.de](http://www.qbasic.de) in der "QB-MonsterFAQ". Hier nun ein paar Hilfestellungen für besonders oft vorkommende Programieraufgaben:

- Suchalgorithmen: Hierfür ist der INSTR-Befehl äußerst nützlich; siehe Onlinehilfe und {11/237}
- Sortieren von Zeichenketten (alphanumerisch) und numerischen Feldern:
  - Bubble Sort: siehe {11/236} und SORT.BAS
  - Quick Sort (rekursiv): siehe {11/241}, {9/294} und SORT.BAS
  - Quick Sort (iterativ): siehe SORT2.BAS
  - Shell Sort: siehe {9/71ff}; {6/281ff} und QuickBASIC 4.5 Hilfe zum SWAP-Befehl



\*\*\*\*\*

\* Literatur zu QBasic (Literaturhinweise: {x/n} = Seitennummer n im Buch {x} )  
\*\*\*\*\*

Über QBasic, QuickBASIC und PowerBASIC (früher "TurboBASIC") gibt es eine  
Riesenauswahl guter Bücher. Die komplette Übersicht finden Sie auf meiner  
Webseite [www.qbasic.de](http://www.qbasic.de) unter "QBasic -> Bücher". Hier nun eine kleine Auswahl  
der im vorliegenden QBasic-Kochbuch zitierten Bücher. Die Bücher {3}, {4}, {5}  
{6} und {9} sind heute noch im Buchhandel oder direkt beim Verlag erhältlich  
(Stand 24.9.05):

- {3} "Das Einsteigerseminar QBASIC" von Heinz-Gerd  
Raymans, bhv-Verlag, 1998  
ISBN 3-89360-672-6, 215 Seiten, 19,80 DM. Erhältlich bei [www.amazon.de](http://www.amazon.de)  
unter dem Suchbegriff "Einsteigerseminar QBasic".
- {4} "Meine 15 schönsten Quick-BASIC Programme" Band 1, Ludwig Schulbuch,  
ISBN 3-929466-58-9, 98 Seiten, 14,80 DM sehr guter QBasic-Kurs für  
Anfänger. Direkt beim Verlag erhältlich bei [www.ludwig-schulbuch.de](http://www.ludwig-schulbuch.de)  
(unter "Programmiersprachen")
- {5} "Meine 15 schönsten Quick-Basic Programme" Band 2, Ludwig-Schulbuch,  
ISBN 3-929466-61-9, 114 Seiten, 14,80 DM sehr guter QBasic-Kurs für  
Fortgeschrittene. Direkt beim Verlag erhältlich bei  
[www.ludwig-schulbuch.de](http://www.ludwig-schulbuch.de) (unter "Programmiersprachen")
- {6} "Arbeiten mit QBASIC" von M Halvorson, Vieweg-Verlag, ISBN 3-528-05164-7,  
520 Seiten, ca. 88,- DM; vollständige, etwas trockene Einführung in  
QBasic
- {9} "Programmierung in Quick BASIC" von N. Kantaris, Bernard Babani Books,  
ISBN 0 85934 229 8, 173 Seiten, englisch, ca. 25,-, sehr gut, ausführ-  
liche Behandlung der Dateizugriffe und von Datenbank-Lösungen, jedoch  
Grafik und Sound nicht behandelt
- {11} "Das QBasic 1.1 Buch" von H.-G. Schumann, Sybex-Verlag, 1993,  
ISBN 3-8155-0081-8, 550 Seiten mit Diskette, 59,- DM sehr gut, behandelt  
fast alle QBasic-Befehle mit vielen Beispielen, beschreibt den Umstieg  
auf QuickBASIC und PowerBASIC; Das Buch ist vergriffen, wird aber  
gelegentlich bei eBay angeboten.

\*\*\*\*\*

\* Liste der Beispielprogramme  
\*\*\*\*\*

Die 48 Beispielprogramme stehen in der Download-Version des QBasic-Kochbuchs im  
Verzeichnis PROGS\ zur Verfügung. Die Download-Version finden Sie auf meiner  
Webseite [www.qbasic.de](http://www.qbasic.de) in der Rubrik "QBasic -> Tutorials". Die  
Beispielprogramme können durch Anklicken der Links "heruntergeladen" bzw.  
abgespeichert und auch direkt gestartet werden, falls die Dateiendung .BAS auf  
dem PC korrekt mit der QBasic-Entwicklungsumgebung QBASIC.EXE verknüpft ist.

- BIOSDAT .BAS = Abfrage des BIOS-Datums mit PEEK
- BOX .BAS = Anzeige eines Kastens mit vielen Gestaltungsmöglichkeiten
- BSAVE1 .BAS = Demonstration der Befehle BSAVE und BLOAD
- CALLREVA .BAS = Unterschied zwischen "Call by Reference" und "CALL by Value"
- CD4 .BAS = Routinen zum Suchen, Sortieren, Einfügen u. Löschen von  
Feld elementen
- DAT-ZEIT .BAS = Datum und Uhrzeit im deutschen Format anzeigen
- DATEIVOR .BAS = Prüfung ob eine Datei vorhanden ist
- DEC2BIN1 .BAS = Zahlenkonvertierung Dezimal -> Binär
- DRIVECHK .BAS = Zeigt die Laufwerksbuchstaben der vorhandenen Laufwerke an
- FILE-BIN .BAS = Bearbeitung von binären Dateien
- FILE-FLD .BAS = Bearbeitung von Direktzugriffs-Dateien mit FIELD-Puffer
- FILE-SEQ .BAS = Bearbeitung von sequentiellen Dateien
- FILE-TYP .BAS = Bearbeitung von Direktzugriffs-Dateien mit TYPE-Puffer
- FILECOPY .BAS = Kopieren einer Datei - byteweise ohne SHELL
- FLDPARAM .BAS = Übergabe von Feldern an SUBs und FUNCTIONS
- GETPUT1 .BAS = Demonstration der Grafik-PUT/GET-Befehle
- GETPUT2 .BAS = Demonstration der Modi des Grafik-PUT-Befehls
- GLOBLFLD .BAS = In SUBs als global deklarierte Felder
- GLOBLVAR .BAS = Variablen in Haupt- und Unterprogrammen gemeinsam verwenden
- JOYINTR .BAS = Joystick ereignisgesteuert mit ON STRIG... abfragen
- JOYTEST .BAS = Joystick-Testprogramm, demonstriert auch den Befehl  
VIEW PRINT
- KAESTEN .BAS = Zeichen von Rechtecken, auch mit runden Ecken
- KLAVIER .BAS = Klavierspielen am PC mit PLAY und SOUND
- MEHRUECK .BAS = Eine SUB/FUNCTION gibt mehr als einen Wert zurück
- MENU1 .BAS = Textbasiertes Auswahlmü
- MENU3 .BAS = Beispiel für ein Auswahlmü mit ON KEY
- MUSE .BAS = Mausprogramm fuer den Grafikmodus
- MOUSEDEU .BAS = Mausroutinen fuer alle SCREENS
- MUSIK .BAS = Diverse Songs und Soundeffekte mit PLAY und SOUND abspielen
- MUSTER .BAS = Zeichnen gemusterter Farbflächen
- ONKEY .BAS = Demonstration des ON KEY... Befehls
- ONTIMER .BAS = Uhrzeitanzeige mit ON TIMER unabhängig vom Hauptprogramm
- OVERFLOW .BAS = Abfangen des Programmabbruchs bei Zahlenüberlauf
- PEEKPOK1 .BAS = Zugriff auf numerische Variablen über PEEK und POKE
- PEEKPOK2 .BAS = Direkter Speicherzugriff auf Strings mit PEEK und POKE
- RANDOMNO .BAS = Zufallszahlen ohne Wiederholung erzeugen, demonstriert  
auch den REDIM-Befehl
- RECURSE .BAS = Eine SUB ruft sich selbst auf ("Rekursion")
- RESTORE .BAS = Demonstriert die Befehle RESTORE, DATA und READ
- RGBFARBE .BAS = Beliebige RGB-Farben erzeugen mit PALETTE
- SCREENRD .BAS = Auslesen des Text-Bildschirms mit SCREEN
- SCREENS .BAS = Testen der Bildschirmmodi
- SEQERROR .BAS = Stellt fest, ob eine sequentielle Datei vorhanden ist
- SINUS .BAS = Koordinatenskaliierung wit WINDOW
- SORT .BAS = Sortierprogramme mit den Methoden Bubble Sort und Quick Sort
- SORT2 .BAS = QuickSort-Algorithmus (nicht rekursiv, sondern iterativ)
- TASTCODE .BAS = Ermittlung des Tastaturcodes beliebiger Tasten
- VARPTR .BAS = Demonstration des VARPTR\$-Befehls
- WEEKDAY .BAS = Ermittlung des Wochentags So...Fr zu einem beliebigen Datum

----- Ende des QBasic-Kochbuchs -----