TBC (TOKIWA BASIC Compiler)
Benutzerhandbuch

Deutsche Übersetzung

R.-Erik Ebert, Berlin, Dezember 2012

Die deutsche Übersetzung basiert auf der von Dr. Genji OKADA besorgten englischen Übersetzung des Originals.

Inhalt

Einleitung

Der Compiler TBC basiert auf der Programmiersprache BASIC (Beginner's All-purpose Symbolic Instruction Code). Zusätzlich zu den traditionellen BASIC-Anweisungen unterscheidet er zwischen Operationen für Integer- und Real-Typen und erlaubt Unterprogramme bzw. Funktionen mit Argumenten. Außerdem wird eine vereinfachte Programmierung ohne die Verwendung von Zeilennummern unterstützt.

Anstatt des üblichen Gleitpunktformats einfacher Genauigkeit wird ein 5-Byte-Format mit einer Genauigkeit von 9,5 Stellen der Mantisse verwendet.

Die Direktübersetzung der Quelltexte in ausführbaren Maschinencode bewirkt eine schnelle Programmausführung

TBC ist Freeware. Gegenüber dem kommerziellen Produkt TKW-86BC fehlen einige Funktionen, wie z.B. das Linken von Objektdateien. Dennoch verfügt TBC über genügend Möglichkeiten, um den Umgang mit dem Computer und einem Compiler kennenzulernen.

1 Grundlagen

Der TBC-Compiler basiert auf Standard-BASIC. Dieser Abschnitt gibt eine Übersicht und Einführung anhand praktischer Beispiele.

1.1 Programmieren und Übersetzen mit TBC

Übersetzen und starten Sie zunächst ein Beispielprogramm. Sie können sich den Inhalt des Programms am Bildschirm mit Hilfe des `type`-Befehls anzeigen lassen:

```
C:\BAS>type test.bas<cr>
```

Dabei ist "C:\BAS>" ein MS-DOS-Prompt und "<cr>" bedeutet das Drücken der ENTER-Taste.

Der Dateiinhalt sollte etwa so aussehen:

```
*** Sample Program ***

for n=0 to 10
  print n;
next n
end
```

Nun übersetzen (kompilieren) wir die Datei.

```
C:\BAS>tbc test<cr>
```

Nach dieser Eingabe sollte folgende Ausschrift am Bildschirm erscheinen:

```
TOKIWA 8086 BASIC Compiler Ver. 5.55 ***Sample***
Copyright (C) 1985-2000 by Genji OKADA

Complete & Saved
```

Wenn die Übersetzung der Quelldatei erfolgreich verlaufen ist, existiert nun eine ausführbare Datei (TEST.EXE). Überprüfen Sie dies mit dem `DIR`-Befehl.

Nun starten Sie das Programm, indem Sie auf der Kommandozeile einfach

```
C:\BAS>test<cr>
```

eingeben. Das Ergebnis sollte wie folgt aussehen:

```
0  1  2  3  4  5  6  7  8  9  10
```

Alle Mitteilungen und Ergebnisse werden am Textbildschirm (der Konsole) ausgegeben.

Wie an diesem Beispiel gezeigt, können Sie sehr einfach Programme kompilieren und ausführen. Weitere Beispielprogramme werden weiter unten vorgestellt.

1.2 Erstellen von Programmen

Zum Erstellen eines Programms wird ein Texteditor benötigt. Der Einfachheit halber wollen wir annehmen, wir hätten einen Texteditor mit dem Namen "TED".
Er wird durch Eingabe des folgenden Befehls gestartet:

```
C:\BAS>ted test1.bas<cr>
```

Nun tippen Sie im Editor das folgende einfache Programm:

```
print "Hello, world!"
end
```

Wenn Sie mit der Eingabe des Programms fertig sind, speichern Sie es auf eine Diskette oder Festplatte. Nun übersetzen Sie den Quelltext in derselben Weise wie im vorherigen Abschnitt.

```
C:\BAS>tbc test1<cr>
```

Falls Sie dabei eine Warnung oder Fehlermeldung, wie z.B. "Syntax error" erhalten haben, prüfen Sie, ob Sie das Programm genau wie oben angegeben geschrieben haben und übersetzen es anschließend noch einmal. Sie sollten nun die Meldung "Complete & Saved" erhalten.

Nach erfolgreicher Übersetzung führen Sie das Programm aus.

```
C:\BAS>test1<cr>
```

Das Ergebnis sollte so aussehen:

```
Hello, world!
```

Dieses Programm entspricht dem traditionellen C-Programm von Kernighan und Ritchie. Vergleichen Sie doch einmal die Größe Ihres ausführbaren Programms mit diesem.

Ein weiteres Beispielprogramm:

```
input x
print "square="; x*x, "square root="; sqr(x)
end
```

Nachdem Sie das Programm wie oben beschrieben eingegeben und kompiliert haben, erhalten Sie nach Eingabe einer Zahl am Prompt "?" das Quadrat und die Quadratwurzel der Zahl.

Möglicherweise werden Sie sich fragen, wo die in einem traditionellen BASIC-Programm üblichen Zeilennummern sind. TBC erfordert keine Angabe von Zeilennummern, verwendet sie jedoch als Referenz-Label. Es ist allerdings auch zulässig, Zeilennummern am Beginn jeder Zeile zu notieren.

2 Anweisungen

2.1 Das Programmformat

2.1.1 Das Zeilenformat

(1) **Zeilen**
Alle Zeilen eines Quelltextes müssen mit einem Wagenrücklauf (0DH), gefolgt von einem Zeilenvorschub (0AH) enden. Die Zeilenlänge darf 255 Zeichen nicht überschreiten.

(2) **Zeichencodes**
Für die Übersetzung wird ASCII-Code verwendet, wobei es keine Unterscheidung zwischen Groß- und Kleinschreibung von Bezeichnern gibt, d. h. Zeichen werden, wenn sie nicht zu einem in einfache Anführungszeichen (Apostrophe) eingeschlossenen String gehören, in Großschreibung umgewandelt.

(3) **Zeilennummern**
Zeilennummern werden nur als Referenzen für Sprünge (Sprungmarken) verwendet. Deshalb können sie beliebig vergeben werden und müssen nicht aufsteigend geordnet sein.
Die Zeilennummern im Hauptprogramm, Unterprogrammen und Funktionen sind jeweils unabhängig voneinander. Somit ist es zulässig, in allen Routinen die gleichen Zeilennummern zu verwenden.

(4) **Mehrfachanweisungen**
Innerhalb einer Zeile können mehrere Anweisungen, jeweils durch einen Doppelpunkt (:) getrennt, notiert werden.

(5) **Reservierte Wörter**
Reservierte Wörter (Schlüsselwörter) bestehen aus einer Folge alphanumerischer Zeichen, wobei nicht zwischen Groß- und Kleinschreibung unterschieden wird. Das erste Zeichen, das auf ein Schlüsselwort folgt, darf kein alphanumerisches Zeichen sein.

(6) **Tabulatoren, Leerzeichen, Leerzeilen**
Es ist möglich, zwischen Wörter eine beliebige Anzahl von Leerzeichen oder Tabulatoren einzufügen. Darüber hinaus können Leerzeilen zur besseren Lesbarkeit des Quelltextes eingefügt werden.

2.1.2 Datentypen

TBC unterstützt drei grundlegende Datentypen: *integer*, *real* und *string*.

Eine Integer-Zahl ist eine vorzeichenbehaftete Ganzzahl mit einer Breite von 2 Byte und einem Wertebereich von -32768 bis 32767. Hexadezimalzahlen werden durch ein vorangestelltes Ampersand (&) gekennzeichnet (z. B. bedeutet &1234 1234H).
In Integer-Ausdrücken können Literale wie 'A' und 'AB' als numerische Werte entsprechend dem ASCII-Code verwendet werden.

Für Real-Zahlen wird eine 5-Byte-Gleitpunktdarstellung verwendet, die eine effektive Genauigkeit von 9,5 Stellen der Mantisse bei einem Wertebereich der Absolutwerte von 2.9387359E-39 bis 1.7014118E+38 erlaubt. Sie besteht aus einem 4 Bytes breiten Mantissenanteil und einem Byte (Bias 80H) für den Exponenten.

Zeichenketten (Strings) werden in doppelte Anführungszeichen eingeschlossen notiert und können höchstens 255 Zeichen lang sein. Intern werden Strings jeweils durch ein Null-Byte abgeschlossen. Eine String-Variable enthält die Adresse (Zeiger) des ersten Zeichens des Strings. Dementsprechend lässt sich eine solche Adresse mit

```
dpeek(loc(A$))
```

ermitteln.

2.1.3 Variablen

Variablen der Typen Integer, Real und String werden wie nachfolgend beschrieben definiert:

Integer-Variablen

Namen von Integer-Variablen bestehen aus alphanumerischen Zeichen sowie dem Unterstrich (_) und müssen mit einem Buchstaben beginnen. Die Deklaration selbst wird mit dem Schlüsselwort `integer` oder `defint` eingeleitet. Die Länge eines Variablennamens ist auf 20 Zeichen begrenzt.

Beispiele:

```
integer a, king, zebra
defint i-n
```

Real-Variablen

Analog zu den Integer-Variablen werden Namen aus alphanumerischen Zeichen sowie dem Unterstrich (_) gebildet und müssen mit einem Buchstaben beginnen.

Eine explizite Deklaration des Typs ist nicht erforderlich, jedoch mit Hilfe des Schlüsselwortes `real` möglich.

Beispiele:

```
a, a0, abcdef, abc56
real epsilon
```

String-Variablen

String-Variablen werden durch Bezeichner aus alphanumerischen Zeichen sowie dem Unterstrich (_), die mit einem Dollar-Symbol ($) abgeschlossen sind, oder mit Hilfe des Schlüsselwortes `character` deklariert.

Beispiele:

```
a$, a0$, abcdef$, abc56$
character myString
```

Array-Variablen (Felder)

Array-Variablen werden durch einen Bezeichner für eine Integer-, Real- oder String-Variable, gefolgt von einer Größenangabe, gebildet. Die Deklaration erfolgt mit Hilfe des Schlüsselwortes `dim`.

Beispiele:

```
integer cntr
dim cntr(100)    ← Deklaration eines Integer-Arrays von 100 Elementen
cntr(1)          ← Zugriff auf das Feldelement mit Index 1
```

Hinweis:

Bei Variablennamen wird nicht zwischen Groß- und Kleinschreibung unterschieden. So verweisen beispielsweise die Bezeichner `abcdef, ABCDEF` and `Abcdef` auf ein- und dieselbe Variable.

2.1.4 Arithmetische Ausdrücke

Ausdrücke bestehen aus Konstanten bzw. Variablen, die durch Operatoren miteinender kombiniert werden. Der TBC-Compiler unterscheidet dabei strikt zwischen Ausdrücken vom Typ Integer und Real.

Beispiele:

```
integer i,j,k              ← Integer-Definitionen
real x,y,z                 ← Real-Definitionen (optional)

k=i+j                      ← Integer-Ausdruck
z=x-y                      ← Real-Ausdruck
```

Es sind die arithmetischen Operatoren +, -, *, /, ^ (power, x hoch y) und mod (Modulus) sowie die logischen Operatoren or, xor und and definiert.

Beispiele:

```
integer i,j,k
k=i*j
k=i/j
k=i mod j
k=i and j
```

Außerdem gibt es sechs relationale Operatoren:

=, >, >=, <, <=, <>

Beispiel:

```
if i>=j then k=1 else k=0
```

2.1.5 String-Ausdrücke

String-Ausdrücke bestehen aus String-Konstanten, String-Variablen und String-Funktionen, die durch den Additionsoperator (+) miteinander verkettet sind.

Beispiele:

```
a$="abcde"
b$=left$(a$,3)+"xyz"
```

Ergänzung:

Auch die relationalen Operatoren

=, >, >=, <, <=, <>

lassen sich mit Strings verwenden. Dies ermöglicht in sehr einfacher Weise Zeichenkettenvergleiche. Verglichen wird dabei zeichenweise nach ASCII-Code, d.h. unter Berücksichtigung der Groß- und Kleinschreibung.

2.1.6 Zuweisungsanweisungen

Der berechnete Wert auf der rechten Seite des Gleichheitszeichens (=), das als Zuweisungsoperator dient, wird der Variablen auf der linken Seite zugewiesen.

Syntax (EBNF).

```
<variable> "=" <arithmetic expression> | <string expression>
```

Dabei ist der Typ des rechtsseitigen Ausdrucks durch den Typ der links vom Zuweisungsoperator stehenden Variablen als Integer, Real oder String bestimmt.

Integer-Ausdrücke

Der rechtsseitige Ausdruck einer Integer-Zuweisung wird als Integer-Ausdruck behandelt.

Beispiel:

```
integer k,m,n
k=m+n
n=dpeek(m)-1
k=m*n
k=m/n
```

Real-Werte auf der rechten Seite werden implizit in Integer-Werte umgewandelt, indem eine Rundung auf die nächste Ganze Zahl erfolgt.

Verwenden Sie die Funktion cint(), um Teilausdrücke vom Real-Typ einzubinden.

Beispiel:

Sei x0=1.3 .

```
integer x
x=x0*4+1                ← zugewiesenes Ergebnis ist 5 (1)
x=cint(x0*4)+1          ← zugewiesenes Ergebnis ist 6 (2)
```

Die Zuweisungen an die Variable x sind Integer-Ausdrücke.

Im Fall (1) wird x0 einfach in einen Integer vom Wert 1 konvertiert. Im Fall 2 ist das Argument der `cint`-Funktion ein Real-Ausdruck (mit dem Ergebnis 5.2). Die Umwandlung erfolgt explizit mit der `cint`-Funktion.

Die drei logischen Operatoren `or`, `xor` und `and` können in Integer-Ausdrücken wie folgt verwendet werden:

```
integer x,y,z
x=y or z
y=z xor x
z=(x+y) and &fff0
```

Auch der Modulus-Operator lässt sich in Integer-Ausdrücken verwenden:

```
integer x,y,z
x=y mod z
```

Die Priorität der arithmetischen und logischen Operatoren ist wie folgt festgelegt:

`mod > /, * > -, + > and, xor, or`
höchste Priorität → niedrigste Priorität

Das Ergebnis eines Integer-Ausdrucks wird im AX-Register des 8086 abgelegt. Der Anfangswert des AX-Registers am Beginn der Auswertung eines Integer-Ausdrucks wird durch die Pseudo-Variable ! repräsentiert.

Real-Ausdrücke

Der rechtsseitige Ausdruck einer Real-Zuweisung wird als Real-Ausdruck behandelt.

Beispiele:

```
x=y+z
d=a+b*c
y=sin(x)+z
```

Im Ausdruck vorkommende Integer-Werte werden vor ihrer Verwendung implizit in Werte vom Real-Typ konvertiert. Mit Hilfe der Funktion `float` ist es möglich, Teilausdrücke vom Integer-Typ zu verwenden.

Beispiel:

```
integer x,y,z
y0=x+y-z                ← Implizite Konvertierung aller Terme nach Real
y0=float(x+y-z)         ← Real-Konvertierung nach der Integer-Operation
x0=float(peek(100)*2)   ← Konvertierung nach Integer-Operation mit
                          Integer-Funktion
```

Hinweis:

Indexangaben von Arrays werden *grundsätzlich* als Integer-Ausdrücke berechnet.

String-Ausdrücke

Der rechtsseitige Ausdruck einer String-Zuweisung wird als String-Ausdruck interpretiert. Findet dabei keine Verkettung statt, so wird nur ein Verweis auf die jeweilige Zeichenkette (Zeiger; keine Kopie davon) zugewiesen.

Beispiele:

```
c$=a$+b$+"abc"      ← Verkettung der Zeichenketten
b$=a$               ← nur Zeigerzuweisung
b$=a$+""            ← erzwungene Kopie durch Anhängen eines Leerstrings
```

Zur direkten Manipulation von Zeichenketten lassen sich die `(d)peek()`-Funktion und die `poke`-Anweisung wie nachfolgend gezeigt verwenden.

Beispiel:

```
a$="XYZ"
poke dpeek(loc(a$)),'A'  ← ersetzt ‚X' in a$ durch ‚A'
```

Zu Beginn der Abarbeitung eines String-Ausdrucks repräsentiert die Pseudo-Variable !$ die Zeichenkette, die durch das BX-Register adressiert wird.

2.2 Anweisungen und deren Syntax

2.2.1 Die Deklaration von Variablen

(1) Die INTEGER-Anweisung

Die Anweisung dient der Deklaration von Integer-Variablen.

Syntax:

```
integer <variable name> [, <variable name>]
```

Dabei ist `variable name` ein Bezeichner aus alphanumerischen Zeichen und dem Unterstrich (_), der mit einem Buchstaben beginnt.

Beispiel:

```
integer cntr, const
dim cntr(100)
```

Bei der Deklaration können Variablen auch unmittelbar initialisiert werden.

Syntax:

```
integer <variable name> <constant> [,<variable name> <constant>]
```

Dabei ist `constant` eine ganzzahlige numerische Konstante.

Beispiel:

```
integer cntr 12, const 25
```

(2) Die REAL-Anweisung

Die Anweisung dient der Deklaration von Integer-Variablen.

Syntax:

```
real <variable name> [, <variable name>]
```

Dabei ist `variable name` ein Bezeichner aus alphanumerischen Zeichen und dem Unterstrich (_), der mit einem Buchstaben beginnt.

Beispiel:

```
real xyz, abc
dim xyz(50,10)
```

Bei der Deklaration können Variablen auch unmittelbar initialisiert werden.

```
Syntax: real <variable name> <constant> [,<variable name> <constant>]
```

Dabei ist `constant` eine ganzzahlige numerische Konstante.

Beispiel:

```
real xyz 1.23, abc 2.56
```

(3) Die CHARACTER-Anweisung

Die Anweisung dient der Deklaration von String-Variablen (Zeichenketten).

Syntax:

```
character <variable name> [, <variable name>]
```

Dabei ist `variable name` ein Bezeichner aus alphanumerischen Zeichen und dem Unterstrich (_), der mit einem Buchstaben beginnt.

Beispiel:

```
character string, test
dim string(10)
```

(4) Deklaration lokaler Variablen

Einfache Variablen, die innerhalb eines Unterprogramms oder einer Funktion deklariert werden, sind lokal zu diesem Unterprogramm oder dieser Funktion.

Beispiel:

```
subroutine test
  integer cntr
  real xyz
```

Einfache Variablen, die außerhalb von Unterprogrammen deklariert werden, haben globale Gültigkeit.

Achtung:

Array-Variablen haben immer einen globalen Gültigkeitsbereich, unabhängig davon, ob sie innerhalb oder außerhalb eines Unterprogramms deklariert werden.

(5) Deklaration von Parametern

Parameterdeklarationen werden für Unterprogramme und Funktionen benötigt.

Beispiel:

```
subroutine test(i, xyz, abc$)
  integer i
  real xyz
```

Dabei müssen grundsätzlich alle Parameter, mit Ausnahme von Zeichenkettenargumenten, deren Name mit dem Dollarzeichen ($) endet, deklariert werden.

2.2.2 Definition von Integer-Konstanten

Integer-Konstanten werden durch einen Bezeichner, der mit einem Unterstrich (_) beginnt, definiert.

Syntax:

```
_<constant name> <integer number>
```

Beispiel:

```
_const    1234
_port     &80
_size     100*2+1          ← Wertfestlegung mit konstantem Ausdruck

integer n, x, y
dim y(_size)               ← Feld mit Konstante als Größenangabe

n=dpeek(_const)+_size      ← Konstante als Argument
x=inp(_port)
```

Solche Konstanten werden bereits vom Compiler verarbeitet. Sie sind in beliebigen ganzzahligen Ausdrücken und auch im Inline-Assembler verwendbar.

2.2.3 Deklaration von Datenfeldern (Arrays)

(1) Die DIM-Anweisung

Die DIM-Anweisung dient der Deklaration von Datenfeldern (Arrays) mit ihrem Namen und der zugehörigen Feldgröße.

Syntax:

```
DIM <array name>(<maximum of index> [,<maximum of index>])
```

Es werden ein- und zweidimensionale Felder unterstützt.

Beispiel:

```
integer a
real x0
dim a(9), x0(20, 30), test$(100)
```

Die Feldnamen sowie Datentypen der Feldelemente werden zunächst wie einfache Variablen deklariert. Mit Hilfe der darauf folgenden DIM-Anweisung erfolgt die Festlegung von Dimension und Größe.

Die Indexzählung beginnt stets mit Null. Für das zehnelementige Integer-Array in obigem Beispiel können somit die einzelnen Feldelemente mit a(0), a(1) … a(9) angesprochen werden.

Die Elemente von zweidimensionalen Feldern werden im Speicher fortlaufend in der Reihenfolge x0(0,0), x0(1,0), x0(2,0) … x0(0,1), x0(1,1) … usw. angelegt.

Zum Ansprechen der Elemente des String-Arrays werden test$(0), test$(1) usw. verwendet. Die Maximallänge der einzelnen Zeichenkettenelemente beträgt standardmäßig 255 Zeichen. In den Feldelementen selbst wird die Adresse (Zeiger) des ersten Zeichens der jeweiligen Zeichenkette abgelegt.

Mit der DIM-Anweisung deklarierte Felder werden im Datensegment angelegt. Sie teilen sich damit den (auf 64 KBytes begrenzten) Speicherplatz mit den einfachen Variablen und Puffern.

Um Speicherplatz zu sparen werden Arrays immer als globale Variablen angelegt.

Ergänzungen:

Die zuvor deklarierten einfachen Variablen mit den Feldern entsprechenden Namen können unabhängig von den Feldern weiter als einfache Variablen gebraucht werden. Für die Definition von Real-Feldern ist die vorherige Definition einer gleichnamigen einfachen Variablen nicht erforderlich.

Einer Variablen wird vom Compiler nur dann Speicherplatz zugewiesen, wenn sie tatsächlich verwendet, d.h. initialisiert wird. Deshalb spart man keinen Speicherplatz, wenn man die Deklaration einer einfachen Real-Variablen als Vorlage für ein Feld weglässt.

Die deklarierte Größe eines Feldes wird nicht im Programm abgelegt. Das bedeutet, dass es keine definierte Möglichkeit gibt, zur Laufzeit die Größe eines Feldes zu bestimmen.

(2) Die DIMX-Anweisung

Die DIMX-Anweisung dient der Deklaration von erweiterten Datenfeldern (extended Arrays) mit ihrem Namen und der zugehörigen Feldgröße.

Syntax:

```
DIMX <array name>(<maximum of index> [,<maximum of index>])
```

Es werden ein- und zweidimensionale erweiterte Felder unterstützt.

Beispiel:

```
integer b
real y0
```

```
dimx b(30000), y0(200, 200)
```

Mit der DIMX-Anweisung deklarierte Felder werden im Extrasegment oberhalb des Stack-Segments angelegt.

Es ist zu beachten, dass die Elemente von mit DIM angelegten Feldern mit 0 initialisiert werden, während mit DIMX angelegte Felder nicht initialisiert werden.

2.2.4 Die DEFINT-Anweisung

Die Anweisung erlaubt die Deklaration einer Gruppe von Integer-Variablen anhand des ersten Buchstabens des Variablennamens.

Syntax:

```
DEFINT <top character of variables> [, <top character > […]]
```

Beispiel:

```
defint i-n, t
```

Hier wird festgelegt, dass für alle Variablen, deren Namen mit den Buchstaben i bis n (in alphabetischer Ordnung) oder t beginnen, implizit der Typ Integer angenommen wird. So würden z.B. die Variablen `jack` und `king()` als einfache bzw. Feldvariablen vom Typ Integer interpretiert werden.

Hinweis:

Die mit DEFINT getroffene Typfestlegung ist „schwächer“ als explizite Typdeklarationen (siehe 2.2.1), das bedeutet, explizit deklarierte Datentypen „gewinnen“ im Fall von Konflikten.

2.2.5 Die BREAK-Anweisung

BREAK-Anweisung dient dem Verlassen des Anweisungsblocks von SWITCH-CASE, FOR-NEXT, WHILE-WEND und REPEAT-UNTIL Anweisungen (vgl. Abschnitt 2.3).

Syntax:

```
BREAK
```

Beispiel:

```
integer n
for n=0 to 10
  if n=5 then break
  print n;
next n
end
```

Als Ergebnis sollte folgende Zahlenfolge ausgegeben werden:

```
0  1  2  3  4
```

2.2.6 Die CONTINUE-Anweisung

Die CONTINUE-Anweisung überspringt der Rest des Anweisungsblocks in FOR-NEXT, WHILE-WEND, REPEAT-UNTIL Schleifen und beginnt den nächsten Schleifendurchlauf.

Beispiel:

```
integer n
for n=0 to 10
  if n=5 then continue
  print n;
next n
end
```

Das Ergebnis sieht wie folgt aus:

```
0  1  2  3  4  6  7  8  9  10
```

2.2.7 Die STOP-Anweisung

Die Anweisung bricht die Ausführung des laufenden Programms ab und kehrt zum Betriebssystem zurück, wobei „end“ auf der Konsole ausgegeben wird.

Syntax:

```
STOP
```

Beispiel:

```
if x<0 then stop
```

Die STOP-Anweisung kann an beliebiger Stelle des Programms (überall dort, wo eine Anweisung notiert werden kann) stehen.

2.2.8 Die END-Anweisung

Mit der END-Anweisung lässt sich das Ende des vom Compiler zu übersetzenden Quelltextes festlegen.

Syntax:

```
END
```

Mit dieser Anweisung wird das Kompilieren abgebrochen.

Hinweis:

Im Hauptprogramm ist diese Anweisung optional. Sie kann z.B. verwendet werden, um hinter dem eigentlichen Quelltext erläuternde Informationen zum Programm zu notieren. In Quellcode-Bibliotheken, die mit der MERGE-Anweisung (siehe 2.2.11) eingebunden werden, ist sie zwingend erforderlich.

Anmerkung:

Soweit sich feststellen ließ, ist obiger Hinweis nicht ganz korrekt. Die END-Anweisung ist auch im Hauptprogramm obligatorisch. Außerdem muss sie mit einem Zeilenumbruch abgeschlossen werden; andernfalls produziert der Compiler einen „Illegal end“-Fehler.

2.2.9 Die REM-Anweisung

Mit der REM-Anweisung wird eine Zeile als Kommentar gekennzeichnet und damit von der Übersetzung ausgeschlossen.

Syntax:

```
REM [<remarks>]
```

Alternativ zum Schlüsselwort REM kann auch ein Sternchen (*) am Zeilenanfang zur Kennzeichnung einer Kommentarzeile verwendet werden.

Beispiel:

```
rem memorandum
* remarks
```

Ein unmittelbar nach einer Zeilennummer notiertes Apostroph (’) ist ebenfalls zur Kennzeichnung einer Kommentarzeile verwendbar.

2.2.10 Die KILL-Anweisung

Mit der KILL-Anweisung wird der gesamte Datenbereich für Zeichenketten gelöscht.

Syntax:

```
KILL
```

Genau genommen werden dabei keine Speicherbereiche überschrieben, sondern es wird lediglich der Pufferzeiger (der auf das Ende des String-Bereichs zeigt) neu initialisiert.

Hinweis:

Man kann Teile der String-Daten löschen, indem man den Zeiger (Adresse 0020H) modifiziert, um damit die automatische Freigabe zu vermeiden. Beispielsweise kann man mit `temp=dpeek(&20)` den Zeiger zwischenspeichern, bevor eine Folge von Anweisungen ausgeführt wird und anschließend mit `dpoke &20,temp` wieder zurücksetzen und damit den inzwischen belegten String-Speicher wieder freigeben.

2.2.11 Die MERGE-Anweisung

Mit MERGE wird eine Quellcode-Bibliothek während der Übersetzung ins Hauptprogramm eingebunden.

Syntax:

```
MERGE <"file name">
```

Ist für den Dateinamen keine Erweiterung angegeben, so wird .LIB angenommen. Alle Quellcode-Bibliotheken müssen mit der END-Anweisung (siehe 2.2.8) abgeschlossen werden. Weitere Anweisungen hinter der MERGE-Anweisung (in derselben Zeile) sind nicht zulässig. Die MERGE-Anweisung darf auch nicht innerhalb von Quellcode-Bibliotheken verwendet werden.

Beispiel:

Die Bibliothek "TEST.LIB" hat folgenden Inhalt:

```
print a
end
```

Hauptprogramm, das "TEST.LIB“ verwendet:

```
a=123
merge "TEST"
end
```

Das äquivalente Programm (das, was der Compiler übersetzt) wäre folgendes;

```
a=123
print a
end
```

2.3 Steueranweisungen

2.3.1 Die FOR-NEXT-Anweisung

Die FOR-NEXT-Anweisung führt eine Folge von Anweisungen mit einer festgelegten Anzahl von Wiederholungen aus.

```
Syntax: FOR <control variable>=<initial> TO <ending> [STEP <increment>]

      <statement>
      [<statement>]
      ...

    NEXT [<control variable>]
```

FOR-NEXT-Schleifen können bis zu einer Tiefe von 15 Ebenen verschachtelt werden.

Beispiel: Die Ausgabe der Zahlen von 0 bis 9 wird zehnmal wiederholt-

```
integer j,k
for j=1 to 10
  print
  for k=0 to 9
    print k;
  next k
next j
end
```

Beispiel: Rückwärts-Zählschleife von 10 bis -10 mit einer Schrittweite von 1

```
integer n
for n=10 to -10 step -1
  print n;
next n
end
```

Anmerkung:

Das Schlüsselwort STEP kann für die Rückwärtszählung (analog zu DOWN TO in Pascal), im Gegensatz zum einfachen inkrementierenden STEP von Standard-BASIC, verwendet werden. Dies ist eine Sonderfunktion zur schnellen Ausführung von Integer-Zählschleifen. Ansonsten verhält sich die FOR-NEXT-Schleife wie die in Standard-BASIC.

Beispiel: Verwendung einer Variablen mit STEP

```
step x                  ← additive Schleife mit Schrittweite x
step -x                 ← subtraktive Schleife mit Schrittweite x
step -(x+y)             ← subtraktive Schleife mit Schrittweite (x+y)
```

Hierbei werden für x und (x+y) positive Werte angenommen.

2.3.2 Die WHILE-WEND-Anweisung

Die WHILE-WEND-Anweisung führt eine Folge von Anweisungen so lange aus, wie die hinter WHILE notierte Bedingung erfüllt ist.

Syntax:

```
WHILE <conditional expression>

 <statement>
 [<statement>]
 ...

WEND
```

Beispiel: Ausgabe der Zahlen von 0 bis 10

```
integer n
n=0
while n<=10
  print n;
  n=n+1
wend
end
```

WHILE-Schleifen können bis zu einer Tiefe von 8 Ebenen verschachtelt werden.

2.3.3 Die REPEAT-UNTIL-Anweisung

Die REPEAT-UNTIL-Anweisung führt eine Folge von Anweisungen so lange aus, bis die hinter UNTIL notierte Bedingung erfüllt ist.

Syntax:

```
REPEAT

  <procedure>

UNTIL <conditional expression>
```

Beispiel: Ausgabe der Zahlen von 0 bis 10

```
integer n
n=0
repeat
  print n;
  n=n+1
until n>10
end
```

Der Inhalt einer REPEAT-UNTIL-Schleife wird immer mindestens einmal ausgeführt.
Eine Verschachtelung ist bis zu 8 Ebenen möglich.

2.3.4 Die IF-THEN-ELSE-Anweisung

Die IF-THEN-ELSE-Anweisung dient der bedingten Abarbeitung von Programmabschnitten. Sie führt, falls der angegebene Bedingungsausdruck den Wert `true` ergibt, eine Anweisung oder Anweisungsfolge *A*, andernfalls (optional) eine Anweisung oder Anweisungsfolge *B* aus.

Syntax 1:

```
IF <conditional expression> THEN <statement A> [ELSE <statement B>]
```

Syntax 2:

```
IF <conditional expression> THEN
        <procedure A>
[ELSE
        <procedure B>]
ENDIF
```

In der Variante 1 ist die Codegröße der Anweisung auf 127 Bytes begrenzt und die Notation erfolgt in einer logischen Zeile.

Beispiel:

```
if a<0 then n=1: print "M" else p=1: print "P"
```

In der Variante 2 wie im traditionellen BASI erweitert und muss mit ENDIF abgeschlossen werden.

Beispiel:

```
if a<0 then
  n=1
  print "M"
else
  p=1
  print "P"
endif
```

Wie im Beispiel erkennbar, muss in der erweiterten IF-Anweisung die erste Zeile mit dem Schlüsselwort THEN enden.

Eine Schachtelung von IF-Anweisungen, wie im folgenden Beispiel, ist ebenfalls zulässig:

```
if a>0 then if b>0 then print "++" else print "+-" else print "-?"
```

Nachfolgende erweiterte Anweisung (nach Syntax 2) führt zum selben Ergebnis:

```
if a>0 then
  if b>0 then
    print "++"
  else
    print "+-"
  endif
else
  print "-?"
endif
```

Für bedingte Sprunganweisungen sind folgende Varianten erlaubt:

```
if a>0 then 200
if a>0 then goto 200
```

Achtung: Logische Operationen im Bedingungsausdruck (wie in der folgenden Anweisung) werden nicht unterstützt:

```
if (a>b) and (c>d) then x=a+c
```

Die Anweisung sollte wie folgt modifiziert werden:

```
if a>b then if c>d then x=a+c.
```

Hinweise:

Der Originaltext ist hier an einigen Stellen etwas missverständlich; deshalb noch ein paar Präzisierungen:

- Die Begrenzung Codegröße für die Variante nach Syntax 1 meint offenbar, dass bei dieser Notation bedingte Sprungbefehle mit 8-Bit-Offset generiert werden. Somit bezieht sich die Begrenzung auf die Länge des generierten Maschinencodes für einen Anweisungsteil. Die erweiterte Form (Syntax 2) verwendet dagegen JMP-Befehle mit NEAR-Offset – hier könnte ein Anweisungsteil also theoretisch bis zu 32 KB lang sein.
- Die fehlende Unterstützung für logische Ausdrücke in der Bedingung gilt auch für die erweiterte Form.
- Genau genommen können in beiden Notationsformen *Anweisungsfolgen* in den Anweisungsteilen hinter THEN bzw. ELSE stehen.
- Beide Notationsformen können auch gemischt werden, z.B.:

  ```
  IF a+1>b THEN if c>d THEN
     x=a+c
  ENDIF
  ```

  bzw.

  ```
  IF a+1>b THEN
     if c>d THEN x=a+c ELSE x=a-c
  ENDIF
  ```

2.3.5 Die SWITCH-CASE-(BREAK-,DEFAULT-,SWEND-)Anweisung

Die Anweisung dient der Flusssteuerung mittels Fallunterscheidung.

Hinter dem Schlüsselwort SWITCH wird ein Ausdruck vom integer- oder string-Typ notiert, anhand von dessen Ergebnis anschließend die Fallunterscheidung (Konstante hinter dem Schlüsselwort CASE) vorgenommen, d.h. die auf die Zeile mit CASE folgende Anweisungsfolge bis zur abschließenden BREAK-Anweisung ausgeführt wird.

Zusätzlich kann ein DEFAULT-Zweig (ohne nachfolgenden Selektionswert) existieren, dessen Inhalt abgearbeitet wird, falls keiner der explizit angegebenen Fälle zutreffend ist.

Syntax 1 (integer-Ausdruck):

```
SWITCH <integer expression>
  CASE <constant>
    <procedure>
    BREAK
  CASE <constant>
    <procedure>
    BREAK
  DEFAULT
    <procedure>
    BREAK
SWEND
```

Beispiel:

```
switch asc(a$)
  case 'A'
    print "by A"
    break

  case 'A'+1
    print "by B"
    break

  default
    print "others"
    break
swend
```

Syntax 2 (string-Ausdruck):

```
SWITCH <string expression>
  CASE <string>
    <procedure>
    BREAK
  CASE <string>
    <procedure>
    BREAK
  DEFAULT
    <procedure>
    BREAK
SWEND
```

Beispiel:

```
switch a$
  case "abc"
    print "by abc"
    break

  case "def"
    print "by def"
    break

  default
    print "others"
    break
swend
```

SWITCH-Anweisungen können bis zu einer Tiefe von 5 Ebenen verschachtelt werden.

Ergänzungen:

- In der Variante 1 kann auch ein Bedingungsausdruck angegeben werden, der einen Real-Wert ergibt. Dieser Wert wird implizit in einen Integer-Wert umgewandelt. Für die Selektionskonstanten können auch *konstante Integer-Ausdrücke* stehen, real-Konstanten sind dagegen nicht zulässig.
  Variante 2 gestattet als Selektionskonstanten *keine Ausdrücke*.
- Der DEFAULT-Zweig muss stets als letzter Zweig des Konstrukts notiert werden.
- Das SWITCH-CASE-Konstrukt hat einen „fall through"-Mechanismus. Das bedeutet, dass jeder Zweig (mit Ausnahme des letzten) normalerweise mit einer BREAK-Anweisung abgeschlossen werden muss. Andernfalls „fällt" die Abarbeitung auf den nächsten Zweig durch, d.h. dessen Anweisungen werden ebenfalls ausgeführt.

2.3.6 Die GOTO-Anweisung

Die GOTO-Anweisung verzweigt zu einer Zeilennummer oder einem Label.

Syntax:

```
GOTO <line number> | <label>
```

Dabei unterscheidet sich die Bedeutung der Zeilennummer von der in traditionellem BASIC. Sie hat hier lediglich die Funktion einer Markierung, auf die verwiesen werden kann.

Zeilennummern sind innerhalb des Hauptprogramms sowie einzelner Unterprogramme bzw. Funktionen jeweils voneinander unabhängig. Deshalb kann auf Zeilennummern innerhalb eines Unterprogramms nicht aus dem Hauptprogramm verwiesen werden und umgekehrt. Folgende Formate sind ebenfalls zulässig:

```
goto @name
goto name
```

Achtung:
Der Compiler meldet keinen Fehler, wenn man mit GOTO zu einem Unterprogramm springt; allerdings führt dies zur Laufzeit zu einem kommentarlosen Programmabbruch.

2.3.7 Die GOSUB-Anweisung

Die GOSUB-Anweisung verzweigt zu einem benannten Unterprogramm über dessen Namen oder einem unbenannten Unterprogramm über eine Zeilennummer oder ein Label.

Syntax:

```
GOSUB <line number> | <label> | <subroutine name>
```

Beispiel:

```
gosub 100
```

Folgende Varianten sind ebenfalls zulässig:

```
gosub @name
gosub name
```

Mit der GOSUB-Anweisung lassen sich nur parameterlose Unterprogramme aufrufen. Für den Aufruf mit Parameterübergabe muss die CALL-Anweisung verwendet werden.

Achtung:
Der Compiler meldet keinen Fehler, wenn man mit GOSUB zu einem "gewöhnlichen Label, also keinem Unterprogramm springt; allerdings kann dies zu einem Laufzeitfehler und damit zum kommentarlosen Programmabbruch führen.

2.3.8 Die RETURN-Anweisung

Mit der RETURN-Anweisung wird ein Unterprogramm verlassen und zum Aufrufer zurückgekehrt.

Syntax:

```
RETURN
```

Die Anweisung wird am Ende oder innerhalb eines Unterprogramms notiert, um das jeweilige Unterprogramm zu verlassen.

Beispiel:

```
    <procedure>

  return
```

Ergänzung:
Auch zum Verlassen benannter (mit SUBROUTINE und SUBEND definierter) Unterprogramme *muss* RETURN *unbeding*t verwendet werden. Andernfalls wird die Routine nicht korrekt verlassen, d.h. die Abarbeitung läuft bei der nächsten Anweisung hinter dem Unterprogramm weiter.

Im Unterschied zum Assemblerbefehl /ret setzt die RETURN-Anweisung den Stack-Pointer auf den Wert des Registers BP und weist ggf. den als Argument angegebenen Rückgabewert dem AX-Register zu (siehe 2.7).

2.3.9 Die ON-GOTO-Anweisung

Die ON-GOTO-Anweisung verzweigt anhand des Ergebnisses eines Bedingungsausdrucks zur angegebenen Programmzeile, die als Zeilennummer oder Label spezifiziert sein kann.

Syntax:

```
ON <integer expression> GOTO <line number1> {, <line number2> }
```

Ergibt der Ausdruck 1, so wird zur ersten Zeile, bei 2 zur zweiten usw. verzweigt für andere Werte, also solche, für die keine Zeile als Ziel angegeben wurde, wird die Abarbeitung bei der folgenden Zeile fortgesetzt.

Beispiel:

```
input a$
on instr("CDEP", a$) goto @calc, @display, @end_prg, @print_out
```

2.3.10 Die ON-GOSUB-Anweisung

Die ON-GOSUB-Anweisung ruft anhand des Ergebnisses eines Bedingungsausdrucks das zugehörige Unterprogramm auf, das über eine Zeilennummer, ein Label oder den Unterprogrammnamen angegeben sein kann. Nach der Rückkehr aus dem Unterprogramm wird die Abarbeitung auf der Zeile fortgeführt, die auf die Zeile mit ON-GOSUB folgt. Entspricht der Rückgabe wert keiner der angegebenen Optionen, so wird direkt zur Ausführung der nächsten Zeile übergegangen.

Syntax:

```
ON <integer expression> GOSUB <line number1> [, <line number2> ---]
```

Dabei wird beim Ergebnis 1 des Ausdrucks zum ersten Unterprogramm, bei 2 zum zweiten usw. verzweigt.

Wie nachfolgend gezeigt, können statt der Zeilennummern auch Labels oder Unterprogrammnamen verwendet werden.

Beispiel:

```
input a$
on instr("CDEP", a$) gosub @calc, @display, @print_out
```

2.3.11 Die ON-ERROR-GOTO (GOSUB-) Anweisung

Die Anweisung legt fest, dass beim Auftreten eines Fehlers zur durch Zeilennummer bzw. Label spezifizierten Zeile verzweigt (GOTO) oder ein Unterprogramm aufgerufen wird (GOSUB).

Syntax:

```
ON ERROR GOTO <line number> | <label>

ON ERROR GOSUB <line number> | <label>
```

Die mit der Anweisung festgelegte Zuordnung bleibt bis zur erneuten Definition einer solchen Zuordnung erhalten.

Ergänzungen:

Zweck des Konstrukts ist die Definition von Routinen zur Fehlerbehandlung.

Die ON-ERROR-Anweisung leitet dabei einen „geschützten Programmabschnitt“ ein. Tritt innerhalb eines solchen Bereichs ein Fehler auf, so wird zur angegebenen Programmstelle verzweigt, wo dann üblicherweise eine Fehlerbehandlung (oder im schlimmsten Fall ein definierter Programmabbruch) durchgeführt wird.

Ein mit ON-ERROR eingeleiteter Bereich gilt bis zur nächsten ON-ERROR-Anweisung. Eine explizite Aufhebung der ON-ERROR-Anweisung existiert nicht. Die Festlegung des Bereichs erfolgt dabei zur Laufzeit, d.h., die ON-ERROR-Anweisung wird nur wirksam, wenn sie tatsächlich ausgeführt wird.

Das Ziel der ON-ERROR-Anweisung (Label oder Zeilennummer) bleibt auch dann global gültig, wenn es sich innerhalb eines mit SUBROUTINE definierten Unterprogramms befindet.

Nach der Rückkehr aus einem mit der ON-ERROR-GOSUB-Anweisung festgelegten Unterprogramm wird die Ausführung bei der auf ON-ERROR-GOSUB folgenden Anweisung fortgesetzt, also (anders als in anderen BASIC-Dialekten) nicht hinter der Anweisung, die den Fehler ausgelöst hat.

Beispiel:

```
real inpvar
subroutine inputerr
    input "Invalid Input - please repeat: ", inpvar
    return
subend

goto start

@defaulthandler
print "Error - program aborted"
stop

@start
on error goto defaulthandler
* any code here
input "Type a number: ", inpvar
on error gosub inputerr
result = sqr(inpvar)
print "Square Root of number is "; result
on error goto defaulthandler
* any code here
end
```

2.3.12 Die ON-EOF-GOTO- (GOSUB-) Anweisung

Die Anweisung legt fest, dass beim Erreichen des Dateiendes bei einer Leseoperation zur durch Zeilennummer bzw. Label spezifizierten Zeile verzweigt (GOTO) oder ein Unterprogramm aufgerufen wird (GOSUB). Die Datei wird dabei automatisch geschlossen.

Syntax:

```
ON EOF GOTO <line number> | <label>

ON EOF GOSUB <line number> | <subroutine>
```

Die mit der Anweisung festgelegte Zuordnung bleibt bis zur erneuten Definition einer solchen Zuordnung erhalten.

Ergänzung:

Das grundsätzliche Verhalten der Anweisung entspricht der ON-ERROR-Anweisung (2.3.11). Der Unterschied besteht lediglich darin, dass hier nicht bei einem Fehler, sondern beim Erreichen des Dateiendes verzweigt wird.

2.3.13 Die ON-KEY-GOTO- (GOSUB-) Anweisung

Die Anweisung verzweigt zur angegebenen Programmzeile (GOTO) oder ruft das angegebene Unterprogramm auf (GOSUB), wenn zum Zeitpunkt der Abarbeitung eine Taste gedrückt ist.

Das Ziel der Verzweigung kann als Zeilennummer oder Label spezifiziert sein. Bei Verwendung von GOSUB ist als Ziel auch der Name eines benannten Unterprogramms (siehe Abschnitt 2.7) zulässig.

Syntax:

```
ON KEY GOTO <line number> or <label>

ON KEY GOSUB <line number> or <label>
```

Bei jeder Ausführung dieser Anweisung wird der Status der Konsole (Standard-Eingabe) überprüft.

Ergänzung:

Nach der Rückkehr aus einem mit ON-KEY-GOSUB aufgerufenen Unterprogramm wird die Abarbeitung bei der auf die ON-KEY-Anweisung folgenden Anweisung fortgesetzt. Dies erlaubt beispielsweise eine zyklische Abfrage der Tastatur, ohne dass dabei der Programmablauf unterbrochen wird.

Beispiel:

```
subroutine inputchr
    c$=inkey$
    print c$;
    return
subend

c$ = ""
while c$<>"q"
  on key gosub inputchr
wend
print ""
end
```

2.3.14 Die CALL-Anweisung

Die Anweisung ruft das angegebene benannte Unterprogramm auf und führt es aus.

Syntax:

```
CALL <subroutine name> [‘(‘<argument> {, <argument>} ‘)’]
```

Dabei können (im Unterschied zur GOSUB-Anweisung, vgl. 2.3.7) zusätzliche Argumente angegeben werden, wobei diese Argumente in Anzahl und Typ und Reihenfolge genau mit den im aufgerufenen Unterprogramm definierten Parametern übereinstimmen müssen (siehe Abschnitt 2.7).

Beispiel:

```
integer x,y
real x0
call test(10, 3.0, "string", x, x0, x$, x-y)
```

In obigem Beispiel haben die Argumente die Typen `integer, real, string, integer, string` und `integer` (Hinweis: DUMMY ist ein reserviertes Wort).

Beachten Sie, dass die CALL-Anweisung in traditionellem BASIC nicht existiert.

Argumente vom Typ `integer` werden als Werte (by value), Argumente der Typen `real` und `string` als Referenzen (by reference) übergeben. Ein String-Argument enthält somit die Startadresse (Offset) der Zeichenkette.

Außerdem können vollständige numerische Arrays als Argumente übergeben werden, wobei hier der Name des Arrays, gefolgt von einem leeren Klammerpaar anzugeben ist.

Beispiel:

```
call name( array1( ), array2( ))
```

Ergänzungen:

Die Forderung nach Übereinstimmung der Argumentanzahl und der Argumenttypen zwischen der Definition und dem Aufruf eines Unterprogramms wird vom Compiler nicht überprüft. Eine Verletzung dieser Forderung kann also zu Fehlern zur Laufzeit führen.

Die Interpretation der Argumente erfolgt grundsätzlich so, wie in der Definition des Unterprogramms festgelegt. Deshalb ist die Definition der Typen der Parameter auch obligatorisch. Werden numerische Felder als Übergabeparameter verwendet, so ist innerhalb des Unterprogramms der Elementtyp zu definieren.

Beispiel:

```
subroutine display(arguments())
  real arguments
  print arguments(0), arguments(1)
  return
subend

real args
dimx args(2)
args(0) = 2
args(1) = 1/3
call display(args())
end
```

Das oben erwähnte reservierte Wort DUMMY ist in der Originaldokumentation nicht erklärt. Es kann als Argument bei der Definition sowie beim Aufruf von Unterprogrammen bzw. Funktionen verwendet werden und hat offenbar den Zweck, ungenutzte Argumente zu kennzeichnen. Intern wird es wie eine benannte Real-Konstante mit dem Wert 0 behandelt.

2.3.15 Die USER-Anweisung

Die Anweisung ruft ein in Maschinensprache geschriebenes Unterprogramm auf.

Syntax

```
USER <memory address> {,<argument> }
```

Argumente	**Register**
1	AX
2	BX, AX
3	CX ,BX, AX
4	DX, CX, BX, AX

Die Inline-Assembler-Anweisung /CALL sollte gegenüber der USER-Anweisung bevorzugt werden.

Anmerkung:

Wegen der Verwendung der Standardregister zur Argumentübergabe müssen die Argumente 16-Bit-Integer-Werte sein.

2.4 Anweisungen zur Dateneingabe

2.4.1 Die INPUT-Anweisung

Die INPUT-Anweisung erlaubt die interaktive Zuweisung von über die Konsole eingegebenen Werten an Variablen.

Syntax:

```
INPUT [<"message";>] <variable> { , <variable> }
```

Die Konsoleneingabe wird entsprechend der Typen der Argumente konvertiert und den als kommaseparierte Liste angegebenen Variablen zugewiesen. Dabei muss die gesamte Dateneingabe auf einmal erfolgen.

Beispiel:

```
10  input "x,y"; x, y
    print "X+Y="; x+y
    goto 10
    end
```

In obigem Beispiel wird nach der Eingabeaufforderung “x,y” ein Fragezeichen (?) ausgegeben. Dies lässt sich unterdrücken, indem man statt des Semikolons hinter der Zeichenkette ein Komma

(,) notiert. Die Werte werden, wahlweise durch Komma oder Leerzeichen getrennt, auf der Konsole eingegeben. Der Abschluss der Eingabe erfolgt mit einem Zeilenumbruch (Enter-Taste)

Die Eingabe von Zeichenketten wird folgendermaßen notiert:

```
10  input a$, b$
    print a$; b$
    goto 10
    end
```

In diesem Beispiel sollte für eine Eingabe a b,c d die Ausgabe a bc d erscheinen, während die Eingabe “a,b” “c,d” die Ausgabe a,bc,d ergibt.

Ergänzung:

Auch bei der Eingabe von Zeichenketten kann bei Bedarf eine Eingabeaufforderung vorangestellt werden.

Werden die eingegebenen Zeichenketten in Anführungszeichen eingeschlossen, so sind das Leerzeichen und das Komma (wie bei numerischen Eingaben) Trennzeichen. Fehlen die Anführungszeichen, so gilt nur das Komma als Trennzeichen, während Leerzeichen mit zur eingegebenen Zeichenkette gehören. Soll eine Zeichenkette ein Komma enthalten, so muss sie in Anführungszeichen eingeschlossen werden. Ein Anführungszeichen innerhalb einer Zeichenkette wird durch die aufeinanderfolgende Notation zweier Anführungszeichen erreicht, wobei die gesamte Zeichenkette nochmals in Anführungszeichen einzuschließen ist.

2.4.2 Die LINE-INPUT-Anweisung

Die Anweisung weist eine vollständige Eingabezeile einer Variablen zu.

Syntax:

```
LINE INPUT [<"message";>] <string variable>
```

Ergänzung:

Die optionale Eingabeaufforderung verhält sich wie bei der einfachen INPUT-Anweisung (2.4.1).

Die Eingabe endet mit einem Zeilenumbruch und kann beliebige druckbare Zeichen enthalten.

Grundsätzlich kann die zur Eingabe verwendete Variable auch numerischen Typs sein. Dies führt allerdings zu einem Fehler, falls die eingegebene Zeichenfolge nicht als Zahl interpretierbar ist oder außerhalb des möglichen Wertebereichs der Variablen liegt.

2.4.3 Die READ-Anweisung

Die READ-Anweisung liest Daten au seiner zuvor geöffneten Datei und weist sie nacheinander den angegebenen Variablen zu.

Syntax:

```
READ <variable> {, <variable> }
```

Die Zielvariablen der Zuweisung können vom Integer, Real- oder String-Typ sein. Die Standard-Dateierweiterung ist .DAT (beachten Sie bitte den Unterschied zwischen der READ-Anweisung und der DATA-Anweisung in traditionellem BASIC).

Beispiel: Lesen einer Datei (vgl. hierzu das Beispielprogramm LNEQ.BAS)

Datei: DTEST.DAT

```
Test data, 1, 2.345
```

Programm:

```
open "DTEST"
read a$,b,c0
print a$
print b,c0
end
```

Tabulatoren und Leerzeichen am Beginn einer Zeichenkette werden innerhalb der Datendatei ignoriert. Wird die Zeichenkette in Anführungszeichen (“) eingeschlossen, so gehören alle dazwischen stehenden Zeichen zur Zeichenkette. Die einzelnen Werte können durch Tabulatoren, Leezeichen oder Kommata voneinander getrennt werden, wobei bei Zeichenketten nur das Komma ein Trennzeichen ist, sofern sie nicht in Anführungszeichen eingeschlossen sind.

Ergänzung:

Die mit READ verwendete Datendatei ist immer eine Textdatei. Hat sie eine andere Erweiterung als .DAT, so muss die Erweiterung im Argument der OPEN-Anweisung mit angegeben werden.

Um innerhalb von Zeichenketten in der Datei Anführungszeichen oder das Komma unterzubringen, gelten die gleichen Regeln wie unter 2.4.1 beschrieben.

Als zusätzliches Trennzeichen ist innerhalb der Datei der Zeilenumbruch zugelassen. D.h., es ist z.B. auch möglich, jeden einzelnen Wert in eine separate Zeile zu schreiben.

Die READ-Anweisung liest bei jedem Aufruf genau so viele Werte aus der Datei, wie Variablen als Argumente angegeben wurden und erwartet, dass auch hinter dem letzten gelesenen Wert ein Trennzeichen steht. Das obige Beispiel führt also zu einem Fehler, wenn hinter dem Wert 2.345 in DTEST.DAT nicht mindestens noch ein Trennzeichen (z.B. ein Zeilenumbruch) steht.

2.4.4 Die LINE-READ-Anweisung

Die Anweisung liest eine Zeile au seiner geöffneten Textdatei und weist deren Inhalt der angegebenen String-Variablen zu.

Syntax:

```
LINE READ <string variable>
```

Beispiel: Die Datendatei ist hier die selbe wie im Beispiel unter 2.4.3.

```
open "DTEST"
line read a$
print a$
end
```

Ergänzung:

Anders als bei der LINE-INPUT-Anweisung, erlaubt die LINE-READ-Anweisung keine numerische Variable als Argument.

Der Aufruf führt zu einem Fehler, wenn die gelesene Zeile nicht mit einem Zeilenumbruch abgeschlossen wurde, was z.B, bei der letzten Zeile einer Datei vorkommen kann. Zum

Abfangen eines solchen Fehlers bietet sich die Verwendung der ON-EOF-GOTO-Anweisung (vgl. 2.3.12) an.

Beispiel: Ausgabe einer Textdatei

```
input "File: ", name$
open name$
* loop forever
while 1>0
  on eof goto end
  line read l$
  print l$
   continue
  @end
   break
wend
end
```

2.4.5 Die READ#-Anweisung

Die Anweisung liest Binärdaten aus einer zuvor mit OPEN# geöffneten Datei und weist sie den als Argumente angegebenen Arrays zu.

Syntax:

```
READ# <array> {, <array> }
```

Nach dem Lesen wird die Datei automatisch geschlossen. Die Anzahl der gelesenen Bytes wird außerdem im Register AX abgelegt.

Beispiel:

```
integer ary
dim ary(25000)
input "File name:", a$
open# a$
read# ary
print !; "bytes"
end
```

Ergänzung:

Als Argumente können Felder (Arrays) mit den Basistypen `integer` und `real` verwendet werden.

Die Ausführung der Anweisung bricht ab, wenn das Dateiende erreicht oder alle als Argumente angegebenen Felder gefüllt sind.

Der im Register AX zurückgegebene Wert ist die Anzahl der Bytes, die in das als letztes Argument angegebene Feld geschrieben wurden.

Die Datei wird am Ende der READ#-Anweisung automatisch geschlossen. Deshalb existiert keine Anweisung, die eine mit OPEN# geöffnete Datei (siehe 2.6.2) explizit schließt.

2.5 Anweisungen zur Datenausgabe

2.5.1 Die PRINT-Anweisung

(1) Die Grundform der PRINT-Anweisung

Die PRINT-Anweisung gibt Ergebnisse der Berechnung eines Ausdrucks aus.

Syntax:

```
PRINT [<expression> {, <expression> }] [, | ;]
```

Beispiel:

```
print 1.2345, 678.9+1.1, "ABCDEF", "test"
print 1.2345; 678.9+1.1; "ABCDEF"; "test"
end
```

Als Ergebnis sollte Folgendes ausgegeben werden:

```
 1.2345          690            ABCDEF  test
 1.2345  690 ABCDEFtest
```

Werden die einzelnen Ausdrücke mit einem Komma getrennt, so wird der Cursor gegenüber dem Anfang der Ausgabe des jeweiligen Ausdrucks um 16 Stellen für Real-Werte bzw. (mindestens) 8 Stellen für Integer- und String-Werte nach rechts versetzt. Verwendet man statt eines Kommas ein Semikolon (;), wird der Cursor nicht um einen festen Betrag verschoben, sondern lediglich Leerzeichen hinter numerischen Ausgaben eingefügt.

Am Ende der PRINT-Anweisung wird ein Zeilenumbruch (Wagenrücklauf und Zeilenvorschub) ausgegeben, sofern die Anweisung nicht mit einem Komma oder Semikolon abgeschlossen wurde.

(2) Festlegung des Ausgabeziels per Index

Das Ausgabeziel wird mit dem #n-Kommando festgelegt.

Syntax:

```
PRINT #<index number> [<expression> {, <expression> }
```

Dabei kann index number Werte von 0 bis 5 annehmen und wird mit #0 initialisiert. Ein einmal festgelegter Ausgabeindex bleibt bis zu einer Neufestlegung gültig.

Die Indizes haben folgende Bedeutung

```
Index          Ziel

#0             Ausgabe auf die Konsole
#1             Ausgabe auf den Standarddrucker
#2 … #4        Ausgabe in Dateien
#5             Serielle Ausgabe (RS-232C, AUX)
```

Beispiel: Druckerausgabe

```
print #1 "test"
print "line out"      ← #1 bleibt als Ausgabeziel erhalten
end
```

Hinweis:
Die Ausgaben der Eingabeaufforderung einer INPUT-Anweisung sowie von Fehlermeldungen sind unabhängig von dieser Festlegung. Sie werden immer auf die Konsole ausgegeben.

(3) Formatsteuerung

Mit dem %-Kommando lässt sich das Format der Ausgebe steuern.

Syntax:

```
PRINT %<format> [<expression> {, <expression> }
```

Es sind folgende Formatangaben möglich:

%n.m Gleitpunktausgabe mit m Dezimalstellen (m<10), und einer gesamten Feldbreite (einschließlich Vorzeichen und ggf. Leerzeichen) von n Stellen. Reicht die angegebene Stellenzahl nicht aus, wird automatisch zur E-Form der Ausgabe gewechselt.

%nEm Gleitpunktausgabe in Expotentialdarstellung (E-Form) mit m Stellen der Mantisse (m<10) und einer Gesamtbreite von n.

%nI Ausgabe als Integer-Wert mit n Stellen.

%I Unformatierte linksbündige Ausgabe als Integer-Wert.

%nH Hexadezimalausgabe eines Integerwertes mit folgender Darstellung in Abhängigkeit von n:

```
n>4        &ABCD
n=4        ABCD
n=3        CD
n=2        CD
```

%nX Ausgabe von n Leerzeichen.

%0 Unformatierte linksbündige Ausgabe als Real-Wert. Dies ist die initiale Standardeinstellung der PRINT-Anweisung.

Beispiel:

```
for n=0 to 10
  x0=n/10
  print %5.1 x0; %12.6 sin(x0); cos(x0)
next n
end
```

Schauen Sie sich das Ergebnis des Beispiels an:

```
0.0    0.000000    1.000000
0.1    0.099833    0.995004
0.2    0.198669    0.980067
0.3    0.295520    0.955336
0.4    0.389418    0.921061
0.5    0.479426    0.877583
0.6    0.564642    0.825336
0.7    0.644218    0.764842
0.8    0.717356    0.696707
0.9    0.783327    0.621610
1.0    0.841471    0.540302
```

Wie Sie sehen, bleibt die zuletzt festgelegte Formatierung erhalten, so dass die Ergebnisse der cos-Funktion wie die der sin-Funktion ausgegeben werden.

Beim freien Format wird der Typ des jeweiligen Ausdrucks aus der ersten darin vorkommenden Zahl, Variablen oder einem Funktions-Rückgabewert bestimmt. Das Kriterium der Typbestimmung ist dabei dasselbe wie beim linksseitigen Ausdruck in den Bedingungen der von IF, WHILE bzw. UNTIL.

Beispiel für die Auswertung als Integer-Ausdruck:

```
print peek(12)+3
```

Der folgende Ausdruck wird als Real ausgewertet:

```
print sin(xyz)+0.1
```

Weitere Beispiele:

```
(1) print 1+sin(0.1)        ergibt 1.0998334 als real-Wert
(2) print 5/3.0             ergibt 1.6666667 als real-Wert
(3) print 5/3               ergibt 1 als integer-Wert
(4) print cint(5/3)         ergibt 2, wobei das Argument ein real-Wert ist
(5) print peek(12)+3.1      führt zu einem Syntaxfehler (gemischte Typen)
```

Verwenden Sie "`print float(peek(12))+3.1`", um den Fehler bei (5) zu vermeiden.

Beispiel für die Tabellenausgabe mit Auswertung von Tabulator-Zeichen:

```
subroutine tabstring(n)
  integer n
  while peek(n)<>0
    if !=9 then print, else print chr$(!);
    n=n+1
  wend
  print
  return
subend
```

Ergänzung:

In obigem Beispiel wird ein kleiner Trick angewendet. Das Unterprogramm tabstring erwartet ein Integer-Argument. Ruft man es mit einer Zeichenkette auf, so wird die Anfangsadresse der Zeichenkette in n übergeben. Nun kann geprüft werden, ob das jeweilige Zeichen ein Tabulator (ASCII-Code 9) oder ein „gewöhnliches" Zeichen ist. Anschließend wird n inkrementiert und damit das nächste Zeichen adressiert. Das Ende ist erreicht, wenn das abschließende Null-Byte der Zeichenkette gelesen wurde.

2.5.2 Die WIDTH-Anweisung

Die Anweisung legt die Zeilenlänge für die PRINT-Anweisung fest.

Syntax:

```
WIDTH #<index number> <line width>
```

Beispiel für die Konsolenausgabe:

```
width #0 72
```

Die Indizes für das Ausgabegerät entsprechen denen der PRINT-Anweisung (2). Dabei kann #0 auch weggelassen werden. Die Zeilenlänge ist für alle Geräte auf 80 voreingestellt (beachten Sie, dass für viele Konsolen 79 die geeignete Einstellung ist).

2.5.3 Die TAB( )-Funktion

Mit TAB wird der Cursor innerhalb der PRINT-Anweisung am die angegebene Position verschoben.

Syntax:

```
TAB(<cursor position>)
```

Beispiel:

```
print "abcdefg"; tab(20); "xyz"
```

Beachten Sie, dass Steuercodes (<20h außer 8 und 0dh) nicht mitgezählt werden.

2.5.4 Die WRITE#-Anweisung

Die Anweisung schreibt Binärdaten aus den angegebenen Arrays in eine vorher geöffnete Datei.

Syntax:

```
WRITE# <array> {, <array> }
```

Nach Abschluss des Schreibvorgangs wird die Datei automatisch geschlossen.

Beispiel für das Schreiben in eine Datei:

```
integer n,x
real y0
dim x(9), y0(9)
for n=0 to 9
  x(n)=n
  y0(n)=n/100
next n

open# "ARYTEST"
write# x, y0
end
```

2.6 Anweisungen zur Dateisteuerung

2.6.1 Die OPEN-Anweisung

OPEN öffnet eine Textdatei zum Lesen mit der READ-Anweisung (vgl. 2.4.3) bzw. LINE-READ-Anweisung (2.4.4).

Syntax:

```
OPEN <"file name">
```

Der angegebene Dateiname kann eine vollständig qualifizierte oder relative Pfadangabe enthalten. Ist keine Dateiendung angegeben, so wird als Standard die Erweiterung ".DAT" angenommen.

2.6.2 Die OPEN#-Anweisung

OPEN# öffnet eine Binärdatei zum Lesen mit der READ#-Anweisung (2.4.5) oder Schreiben mit der WRITE#-Anweisung (2.5.4).

Syntax:

```
OPEN# <"file name">
```

Der angegebene Dateiname kann eine vollständig qualifizierte oder relative Pfadangabe enthalten. Ist keine Dateiendung angegeben, so wird als Standard die Erweiterung ".ARY" angenommen.

2.6.3 Die OPEN#2- … OPEN#4-Anweisungen

Die Anweisungen öffnen eine Textdatei für die Ausgabe mit PRINT#2 … PRINT#4 (siehe 2.5.1).

Syntax:

```
OPEN#2[A] <"file name">
OPEN#3[A] <"file name">
OPEN#4[A] <"file name">
```

Der angegebene Dateiname kann eine vollständig qualifizierte oder relative Pfadangabe enthalten. Ist keine Dateiendung angegeben, so wird als Standard die Erweiterung ".DAT" angenommen.

Wird hinter dem Geräteindex ein “A” anfügt, so werden die ausgegebenen Daten an eine vorhandene Datei angehängt, andernfalls wird der Inhalt überschrieben. Existiert die Datei noch nicht, wird sie automatisch angelegt.

2.6.4 Die Anweisungen CLOSE und CLOSE#2 … CLOSE#4

(1) CLOSE

Die Anweisung schließt eine Datei, die zuvor mit der OPEN-Anweisung zum Lesen mit READ geöffnet wurde.

Syntax:

```
CLOSE
```

Eine Datei wird beim Lesen mit READ automatisch geschlossen, wenn das Dateiende (EOF) erreicht wurde. In diesem Fall ist der anschließende Aufruf von CLOSE nicht erforderlich.

(2) CLOSE#2 … CLOSE#4

Die Anweisungen schließen eine Datei, die zuvor mit der OPEN#2 … OPEN#4 -Anweisung zur Datenausgabe mit PRINT#2 … PRINT#4 geöffnet wurde.

Syntax:

```
CLOSE#2
CLOSE#3
CLOSE#4
```

2.7 Die SUBROUTINE-Anweisung

Die Anweisung SUBROUTINE kennzeichnet den Beginn eines benannten Unterprogramms und registriert dessen Namen.

Syntax:

```
SUBROUTINE <name> [(<parameter> {, <parameter> })]

  <procedure>

  RETURN [<integer expression>]
SUBEND
```

Der Name eines Unterprogramms besteht aus alphanumerischen Zeichen sowie dem Unterstrich (_), wobei das erste Zeichen ein Buchstabe sein muss. Die Maximallänge des Namens beträgt 20 Zeichen, die Anzahl der Parameter höchstens 62.

Am Anfang des Rumpfes des Unterprogramms müssen Integer- und Real-Parameter in der Reihenfolge ihres Auftretens im Prozedurkopf mit Hilfe von INTEGER- und REAL-Anweisungen deklariert werden.

Ergänzung:

String-Parameter müssen (und können) nur dann (mit character) deklariert werden, wenn kein Name mit abschließenden $-Zeichen verwendet wird.

Der optionale Rückgabewert setzt beim Rücksprung aus dem Unterprogramm den Wert des AX-Registers. Wird er weggelassen, ist der Wert zufällig bzw. abhängig von der zuletzt ausgeführten Anweisung.

Die Rückgabe eines Wertes funktioniert in gleicher Weise auch mit unbenannten Unterprogrammen, die mit GOSUB aufgerufen werden.

Ausführlicher wird die Parameterübergabe in der Ergänzung zu Abschnitt 1 behandelt.

Beispiel:

```
integer a
real b0
a=2
b0=1/3
call display(a, b0)
stop

subroutine display(x, yz)
  integer x
  real yz
  print x, yz
  return
subend
end
```

Argumente des Typs Integer werden als Wert übergeben, Real-Argumente per Referenz – siehe hierzu die CALL-Anweisung (Abschnitt 2.3.14).

Beispiel:

```
integer a
real b0
a=12
b0=5.678
call test(a, b0)
print a, b0
stop

subroutine test(x, y0)
  integer x
  real y0
  print x, y0
  x=34
  y0=1.234
  print x, y0
  return
subend
end
```

Das Ergebnis sollte wie folgt aussehen:

```
  12       5.678
  34       1.234
  12       1.234
```

Um eine Integer-Variable per Referenz zu übergeben, also den Inhalt eines Integer-Arguments zu modifizieren, ist folgende Notation möglich:

```
integer a
a=12
call test(loc(a))
print a
stop

subroutine test(x)
  integer x
  print %i dpeek(x)
  dpoke x, 56
  return
subend
end
```

Anmerkung:

Hier wird nicht der Integer-Wert selbst an das Unterprogramm test übergeben, sondern die Adresse der Variablen a, in der der Wert gespeichert ist.

Das Unterprogramm modifiziert dann mit der DPOKE-Anweisung (vgl. 2.9.7) den Wert der Variablen.

Beispiel für numerische Array-Argumente::

```
subroutine test(a(10), b0(5,5))
* subroutine test(a(), b0(,)) ← will work too
  integer a
  real b0
  integer i,j
for i=0 to 10
  print a(i);
next i
print
for i=0 to 5
  for j=0 to 5
     print b0(i,j);
  next j
  print
next i
  return
subend

integer x,i,j
real y0
dim x(10), y0(5,5)
for i=0 to 10
  x(i) = i
next i
for i=0 to 5
  for j=0 to 5
     y0(i,j) = i*j
  next j
next i
call test(x(), y0())
end
```

2.8 Die FUNCTION-Anweisung

Die FUNCTION-Anweisung kennzeichnet den Beginn der Definition einer benutzerdefinierten Funktion und registriert deren Namen.

Syntax:

```
FUNCTION <name> [(<parameter> {, <parameter> })]

  <procedure>

RETURN <real expression>
FNEND
```

Benutzerdefinierte Funktionen sind benannten Unterprogrammen sehr ähnlich. Der Wesentliche Unterschied besteht darin, dass Funktionen stets einen Rückgabewert haben und dieser Wert vom Typ Real ist.

Beispiel für eine Funktion, die ihr Argument mit 1000 multipliziert:

```
real a0
a0=2.0
print fn_sen(a0)
stop

function sen(yz)
  real yz
  return yz*1000.0
fnend
end
```

Anmerkung:

Wird innerhalb einer Funktion ein Integer-Wert als Rückgabewert angegeben, so erfolgt eine automatische Konvertierung nach Real.

Ergänzung:

Die Rückgabe des Funktionswertes erfolgt über die Register CX, DX und AL. Dabei werden die 5 Bytes des Real-Wertes wie folgt auf die (Halb-)Register abgebildet:

Byte0	Byte1	Byte2	Byte3	Byte4
CL	CH	DL	DH	AL

2.9 Sonstige Anweisungen

2.9.1 Die SWAP-Anweisung

Die Anweisung vertauscht variable1 und variable2.

Syntax:

```
SWAP <variable1>, <variable2>
```

Es gibt einige Einschränkungen für das Vertauschen, beispielsweise dann, wenn eine einfache Variable oder ein normales Array für ein erweitertes Array platziert würde.

Ergänzung:

Die etwas ungenaue Beschreibung von Einschränkungen im Original-Handbuch lässt sich folgendermaßen präzisieren:

Streng genommen werden nicht Variablen, sondern Variablenwerte vertauscht. Daraus ergibt sich, dass die als Argumente angegebenen Variablen vom selben Typ sein müssen, andernfalls erhält man schon vom Compiler eine Fehlermeldung.

Gibt man Array-Variablen als Argumente an (unabhängig davon, ob mit DIM oder DIMX definierte), so akzeptiert das der Compiler, die SWAP-Anweisung bleibt aber wirkungslos. Dagegen ist es problemlos möglich, die Werte von Feldelementen mit SWAP zu vertauschen (wieder gleiche Typen vorausgesetzt).

2.9.2 Die WAIT-Anweisung

Die WAIT-Anweisung hält die Programmabarbeitung an, bis die am festgelegten Port gelesenen Daten dem angegebenen Bedingungsausdruck genügen (d.h. der gelesene Wert dem des Ausdrucks entspricht).

Syntax:

```
WAIT <port index>, <integer expression>
```

Beispiel:

```
wait &12, &80
```

Folgende Assembler-Routine entspricht obiger Anweisung:

```
loop0: in    al,12h
       and   al,80h
       jz    loop0
```

2.9.3 Die OUT-Anweisung

Die Anweisung schreibt ein Byte (8 Bits) in den angegebenen Port.

Syntax:

```
OUT <port index>, <integer expression>
```

2.9.4 Die DOUT-Anweisung

Die Anweisung schreibt ein 16-Bit-Wort in den angegebenen Port.

Syntax:

```
DOUT <port index>, <integer expression>
```

2.9.5 Die POKE-Anweisung

Die Anweisung schreibt ein Byte (8 Bits) an die angegebene Speicheradresse.

Syntax:

```
POKE [[<integer variable>];] <memory address>, <integer expression>
```

Beispiel für das Schreiben ins Datensegment:

```
poke &a000, 3
```

Beispiel für die Verwendung des Extrasegments:

```
x=&a000                 ← eseg-Adresse
poke x; address, data   ← zeigt in eseg
```

Wenn der Wert des Extrasegments bereits gesetzt wurde, kann die verkürzte Form "`poke; address, data`" verwendet werden.

Anmerkung:

In der Standardform (erste Variante) wird die angegebene Adresse als Offset im Datensegment interpretiert.

Die zweite Variante erlaubt die explizite Angabe einer Segmentadresse, mit der intern offenbar das ES-Register belegt wird. Zwischen mehreren aufeinanderfolgenden POKE-Aufrufen bleibt diese Belegung erhalten, so dass dann die verkürzte Form (nu rein Semikolon hinter POKE) verwendet werden kann.

2.9.6 Die RANDOMIZE -Anweisung

Die Anweisung initialisiert den internen Zufallszahlengenerator, d.h. sie setzt den Startwert für die RND()-Funktion.

Syntax:

```
RANDOMIZE [<integer expression>]
```

Beispiel:

```
randomize &5a5a
```

Anmerkung:

Die Initialisierung des Zufallszahlengenerators hat vor allem die Aufgabe zu gewährleisten, dass sich die mit RND() erzeugten Zufallsfolgen zwischen mehreren Programmläufen unterscheiden. Deshalb ist es zweckmäßig. für die Initialisierung keine Konstante, sondern einen Wert zu verwenden, der sich von Programmstart zu Programmstart ändert, beispielsweise die Systemzeit.

2.9.7 Die DPOKE-Anweisung

Die Anweisung schreibt ein Wort (16 Bits) an die angegebene Speicheradresse.

Usage:

```
DPOKE [[<integer variable>];] <memory address>, <integer expression>
```

Mit Ausnahme der Datengröße entspricht das Verhalten genau dem von POKE (siehe 2.9.5).

2.9.8 Die VALUE-Anweisung

Die VALUE-Anweisung legt einen Integer-Wert im AX-Register ab.

Syntax:

```
VALUE <integer expression>
```

Der Wert (das Ergebnis des Integer-Ausdrucks) verbleibt im AX-Register, ohne dass eine Zuweisung erfolgt. Eine genauere Darstellung mit Beispielen findet sich in Abschnitt 3.4.

2.9.9 Der Inline-Assembler

TBC gestattet die inline-Verwendung von Assembler-Anweisungen.

Syntax:

```
/mnemonic
```

Beispiel:

```
/mov    ax,bx
/add    ax,dx
/push   ax
```

Die folgende Form ist ebenfalls erlaubt:

```
/mov ax,bx  /add ax,dx  /push ax
```

Unterstützte Assembler-Anweisungen:

```
Mnemonic            Action

  aaa               ;ASCII adjust for add
  aad               ;ASCII adjust for divide
  aam               ;ASCII adjust for multiply
  aas               ;ASCII adjust for subtract
  adc               ;add with carry
  add               ;add
  and               ;logical and
  call              ;call procedure
  cbw               ;convert byte to word
  clc               ;clear carry flag
  cld               ;clear direction flag
  cli               ;clear interrupt enable flag
  cmc               ;complement carry flag
  cmp               ;compare
  cmpsb             ;compare string(byte)
  cmpsw             ;compare string(word)
  cwd               ;convert word to double word
```

```
daa             ;decimal adjust for addition
das             ;decimal adjust for subtraction
dec             ;decrement
div             ;divide
hlt             ;halt
idiv            ;integer divide(signed)
imul            ;integer multiply(signed)
in              ;input from port
inc             ;increment
int             ;interrupt
into            ;interrupt on overflow
iret            ;interrupt return
ja              ;jump on above
jb              ;jump on below
jbe             ;jump on below or equal
jc              ;jump on carry
jcxz            ;jump on CX zero
jg              ;jump on greater
jge             ;jump on greater or equal
jl              ;jump on less
jle             ;jump on less or equal
jmp             ;jump unconditionally
jmps            ;jump short unconditionally
jnb             ;jump on not below
jnc             ;jump on carry
jno             ;jump on no overflow
jpo             ;jump on parity odd
jns             ;jump on no sign
jnz             ;jump on not zero
jo              ;jump on overflow
jpe             ;jump on parity even
js              ;jump on sign
jz              ;jump on zero
lahf            ;load AH into flag
lds             ;load pointer to DS
lea             ;load EA to register
les             ;load pointer to ES
lock            ;lock bus
lodsb           ;load string(byte)
lodsw           ;load string(word)
loop            ;loop CX times
loopz           ;loop while zero
loopnz          ;loop while not zero
mov             ;move to and from registers/memory
movsb           ;move string(byte)
movsw           ;move string(word)
mul             ;multiply(unsigned)
neg             ;change sign
nop             ;no operation
not             ;invert register/memory
or              ;logical or
out             ;output to port
pop             ;pop a word from the stack
popf            ;pop flags
push            ;push operand onto the stack
pushf           ;push flags
rcl             ;rotate left through carry
rcr             ;rotate right through carry
rep             ;repeat if equal
rep             ;repeat if zero
repnz           ;repeat if not zero
ret             ;return
retf            ;return far
rol             ;rotate left
```

```
    ror             ;rotate right
    sahf            ;store AH into flags
    sar             ;shift arithmetic right
    sbb             ;subtract with carry
    scasb           ;scan string(byte)
    scasw           ;scan string(word)
    shl             ;shift left
    shr             ;shift right
    stc             ;set carry flag
    std             ;set direction flag
    sti             ;set interrupt enable flag
    stosb           ;store string(byte)
    stosw           ;store string(word)
    sub             ;subtract
    test            ;test
    wait            ;wait
    xchg            ;exchange
    xlat            ;translate string
    xor             ;exclusive or
```

The others:

```
    cs              ;code segment override prefix
    ds              ;data segment override prefix
    es              ;extra segment override prefix
    ss              ;stack segment override prefix
    byte            ;define byte
    word            ;define word
    float           ;floating point number
    even            ;even directive
    align           ;align directive(4,16,256)
    blank           ;relative org without initialize
    byte ptr        ;byte pointer operator override
    word ptr        ;word pointer operator override
    offset          ;offset operator of address
```

Ergänzung:

Der Inline-Assembler unterstützt nicht alle gebräuchlichen Mnemonics. Insbesondere fehlen ein paar (allerdings redundante) Befehle für bedingte Sprunganweisungen sowie Möglichkeiten zur Definition vom 32-Bit-Werten und für FAR-Aufrufe (JMP und CALL). Hier muss man sich bei Bedarf „zu Fuß“ behelfen, d.h. mit der /byte-Anweisung und handcodierter Maschinensprache.

2.9.10 Konsolenkommandos

Alle Anweisungen dieses Abschnitts setzen einen installierten ANSI.SYS-Treiber voraus.

(1) Die LOCATE-Anweisung

LOCATE erlaubt die freie Positionierung des Cursors auf dem Display.

Syntax:

```
LOCATE <column>, <line>
```

Beispiel:

```
locate x, y
```

wobei x die Spaltennummer (0<=x<80) und y die Zeilennummer (0<=y<25) ist.

(2) Die TXCLS-Anweisung

Die Anweisung löscht den gesamten Text der Anzeige.

Syntax:

```
TXCLS
```

(3) Die BEEP-Anweisung

Die Anweisung gibt einen kurzen Piepton aus.

Syntax:

```
BEEP
```

(4) Die CURSOR-Anweisung

Mit dieser Anweisung lässt sich der Cursor ein- bzw. ausschalten.

Syntax:

```
CURSOR <integer expression>
```

Dabei steht ein Wert von 0 für Aus, ein anderer Wert für Ein.

(5) Die RDELETE-Anweisung

Die Anweisung löscht alle Zeichen bis zum Zeilenende.

Syntax:

```
RDELETE
```

(6) Die TXCOLOR-Anweisung

Die Anweisung legt die Farbe für die Textausgabe fest..

Syntax:

```
TXCOLOR <integer expression>
```

Es gelten folgende Zuordnungen von Werten zu Farben:

```
0=black, 1=blue, 2=green, 3=sky blue, 4=red, 5=magenta,
```

```
6=yellow, 7=white
```

2.10 Eingebaute Funktionen

Anmerkung:

Wie auch in anderen (älteren) BASIC-Dialekten werden eingebaute Funktionen direkt, d.h. im Unterschied zu benutzerdefinierten Funktionen (vgl. 2.8) ohne vorangestelltes FN_ aufgerufen.

Funktionsaufrufe werden außerdem immer als Bestandteil eines Ausdrucks bzw. Rechtswert einer Zuweisung verwendet.

2.10.1 Integer-Funktionen

Diese Funktionen sind (nur) in Integer-Ausdrücken verwendbar.

(1) Die IABS()-Funktion

Die Funktion liefert den Absolutwert (Betrag) eines Integer-Ausdrucks.

Syntax:

```
IABS(<integer expression>)
```

(2) Die IVAL() -Funktion

Die Funktion wandelt eine Zeichenkette in einen Integer-Wert um..

Syntax:

```
IVAL(<string>)
```

Ergänzung:

Die IVAL-Funktion interpretiert Zahlen in Dezimal- und Hexadezimaldarstellung (mit vorangestelltem &).

Ist die übergebene Zeichenkette nicht (oder nicht vollständig) als Integer-Zahl interpretierbar, so wird kein Fehler ausgelöst, sondern es werden – vom Anfang beginnend - so viele Zeichen interpretiert, wie interpretierbar sind. Beginnt die Zeichenkette mit einem für Zahlen ungültigen Zeichen, so ist das Ergebnis 0. Führende Leerzeichen werden ignoriert.

Beispiel:

```
PRINT IVAL("23"), IVAL(" 23"),IVAL("-23"), IVAL("&23"), IVAL("-&23"), IVAL("2.9")
```

ergibt

```
 23      23     -23      35      0       2
```

(3) Die INP()-Funktion

Die Funktion liest ein Byte vom angegebenen Port und liefert dessen Wert als Ergebnis.

Syntax:

```
INP(<integer expression>)
```

Der Wert wird im Low-Byte des Ergebnisses abgelegt und das High-Byte mit Null aufgefüllt..

(4) Die DINP()-Funktion

Die Funktion liest ein Wort (2 Bytes) vom angegebenen Port.

Syntax:

```
DINP(<integer expression>)
```

Anmerkung:

Genau genommen wird das Low-Byte des Ergebnisses vom angegebenen Port und das High-Byte vom darauffolgenden Port gelesen.

(5) Die PEEK()-Funktion

Die Funktion liest ein Byte von der angegebenen Speicheradresse.

Syntax:

```
PEEK([[<integer variable>];]<integer expression>)
```

Beispiel:

```
integer x,address
x=peek(address)
```

Beispiel für das Lesen aus einem anderen Segment:

```
integer x,y,address
x=&a000                 ← eseg-Adresse
y=peek(x; address)      ← zeigt in eseg
```

Anmerkung:

Für die Adressierung von Daten mit Segmentangabe (lies: über FAR-Zeiger) gelten die gleichen Regeln, wie in Abschnitt 2.9.5 für POKE beschrieben.

(6) Die DPEEK()-Funktion

Die Funktion liest ein Wort (2 bytes) von der angegebenen Speicheradresse.

Syntax:

```
DPEEK([[<integer variable>];] <integer expression>)
```

Mit Ausnahme der Breite der gelesenen Daten verhält sich DPEEK() wie die PEEK()-Funktion..

(7) Die POS()-Funktion

Die Funktion gibt die Spaltenposition des Cursors zurück.

Syntax:

```
POS([<integer expression>])
```

Mit dem Integer-Wert wird die Indexnummer festgelegt; verwenden Sie POS() ohne Indexnummer innerhalb der PRINT-Anweisung.

Anmerkung:

Die wirkliche Bedeutung der Indexnummer ließ sich nicht ergründen. Möglicherweise handelt es sich dabei um den Index der Bildschirmseite, auf die sich die Anweisung bezieht. Offensichtlich liefern jedenfalls die Aufrufe POS(0) und POS() identische Ergebnisse.

(8) Die LOC()-Funktion

Die Funktion liefert die Speicheradresse der angegebenen Variablen.

Syntax:

```
LOC(<variable>)
```

Ein Spezialfall ist dabei die Übergabe eines erweiterten (mit DIMX definierten) Arrays ohne Indexangabe als Argument. In diesem Fall liefert die Funktion die Position der Segmentadresse des Arrays, also des Wertes, der dem Extrasegment vor dem Zugriff zugewiesen werden muss.

Beispiel zur Bestimmung der Segmentadresse eines erweiterten Arrays:

```
dpeek(loc(exarray( )))
```

Anmerkung:

Für jede (globale) Variable wird Speicherplatz in Datensegment belegt, dessen Offset-Adresse mit der LOC-Funktion bestimmt werden kann. Mit DIMX definierte (erweiterte) Arrays beginnen immer an Paragraphen (durch 16 teilbare Adressen), also an Segmentgrenzen. Die Segmentadresse des erweiterten Arrays wird im Datensegment gespeichert.

Mit DIM definierte Arrays liegen dagegen direkt im Datensegment. Die Startadresse eines solchen Arrays ist die seines ersten Elements.

(9) Die CINT()-Funktion

Die Funktion rundet den Wert eines Real-Ausdrucks auf den nächsten Integer-Wert.

Syntax:

```
CINT(<real expression>)
```

Beispiel;

```
print cint(2.3), cint(2.5), cint(3.5), cint(2.7), cint(-2.7)
```

ergibt

```
 2        3        4        3       -3
```

(10) Die ASC()-Funktion

Die Funktion liefert den ASCII-Code des ersten Zeichens einer Zeichenkette.

Syntax:

```
ASC(<string>)
```

Beispiel:

```
ASC("abcd")
```

ergibt 61h

(11) Die INSTR()-Funktion

Die Funktion gibt 1 zurück, wenn string2 in string 1 enthalten ist, andernfalls 0.

Syntax:

```
INSTR([<integer expression>,] <string1>, <string2>)
```

Das Ergebnis des optionalen Integer-Ausdrucks bestimmt die Zeichenposition in string1, an der der Vergleich beginnen soll. Der Standardwert ist 0.

Beispiel:

```
integer n, x
n=2
a$="abcdefg"
x=instr(n,a$,"de")
```

Das Ergebnis sollte hier 4 lauten..

(12) Die LEN()-Funktion

Die LEN-Funktion liefert die Anzahl der Zeichen in einer Zeichenkette..

Syntax:

```
LEN(<string>)
```

Anmerkung:

Die Funktion verhält sich wie die strlen-Funktion in C, d.h. sie zählt die Zeichen vom Beginn der Zeichenkette bis zum ersten Auftreten eines Null-Bytes, das als Endekennzeichen interpretiert wird.

(13) Die FRE()-Funktion

Die Funktion liefert die Größe des String-Bereiches nach ausgeführter Garbage-Collection.

Syntax:

```
FRE(<dummy variable name>)
```

Ergänzung:

Die Funktion gibt zunächst nicht mehr benötigte String-Bereiche frei (Garbage-Collection) und liefert als Ergebnis die verbleibende Anzahl freier Bytes im Datensegment.

(14) Die USR()-Funktion

Die Funktion ruft ein in Maschinensprache geschriebenes Unterprogramm auf und gibt den Wert des AX-Registers als Ergebnis zurück.

Syntax:

```
USR(<memory address> {,<argument> })
```

Argumente	Register
1	AX
2	BX, AX
3	CX, BX, AX
4	DX, CX, BX, AX

Ergänzung:

Die Funktion führt einen NEAR-Aufruf des Unterprogramms aus, wobei zuvor die Register wie oben angegeben mit den Werten der übergebenen Integer-Argumente initialisiert werden.

Mit Ausnahme der Aufrufsyntax und der Verwendbarkeit als Rechtswert einer Zuweisung entspricht das Verhalten dem der USER-Anweisung (vgl. 2.3.15)

Beispiel:

```
integer adr
goto start
@usrproc
/mov ax, bx
/ret

@start
/mov ax, @usrproc
adr = !
print usr(adr, 2,1)
user adr, 2,1
print !
end
```

(15) Benutzerdefinierter CALL_name()-Funktionsaufruf

Dieses Konstrukt ermöglicht es, ein mit der SUBROUTINE-Anweisung (siehe 2.7) definiertes Unterprogramm wie eine Funktion aufzurufen, d.h. seinen Rückgabewert (Typ Integer) als Rechtswert einer Zuweisung zu verwenden.

Syntax:

```
CALL_<subroutine name> [(<argument> {, <argument>})]
```

Dazu wird das Präfix CALL_ dem Namen des Unterprogramms vorangestellt.

Beispiel:

```
x=call_name(y)
stop

subroutine name(y)
  integer y

  <procedure>

  return y+3
subend
```

Der Ausdruck hinter RETURN wird als Integer ausgewertet.

2.10.2 Real-Funktionen

Diese Funktionen sind (nur) in Real-Ausdrücken verwendbar.

(1) Die ABS()-Funktion

Die Funktion liefert den Absolotbetrag eines real-Ausdrucks.

Syntax:

```
ABS(<real expression>)
```

(2) Die SGN()-Funktion

Die Funktion liefert den Vorzeichenwert des Ausdrucks.

Syntax:

```
SGN(<real expression>)
```

Der Rückgabewert ist 1 für positive Werte, 0 for Null und -1 für negative Werte.

Anmerkung:

Obwohl der Rückgabewert nur die Werte 1, 0 und -1 annehmen kann, ist er von Typ Real.

(3) Die RND()-Funktion

Die Funktion liefert eine Pseudo-Zufallszahl zwischen 0 und 1.

Syntax:

```
RND(<dummy variable name>)
```

Anmerkung:

Um nicht immer dieselbe Folge von Zufallszahlen zu erhalten, sollte der Zufallszahlengenerator mit der RANDOMIZE-Anweisung (vgl. 2.9.6) initialisiert werden.

(4) Die INT()-Funktion

Die Funktion liefert den größten ganzzahligen Wert, der nicht größer als der Wert des Arguments ist.

Syntax:

```
INT(<real expression>)
```

Beispiel:

```
INT( 2.3)   ← ergibt 2.0
INT(-2.3)   ← ergibt -3.0
```

Anmerkung:

Im Unterschied zur CINT()-Funktion (10) wird hier keine “echte” Rundung durchgeführt. Außerdem ist das Ergebnis vom Typ Real (nicht Integer).

(5) Die FIX()-Funktion

Die Funktion liefert den ganzzahligen Anteil eines Real-Wertes.

Syntax:

```
FIX(<real expression>)
```

Beispiel:

```
FIX(-2.3)   ← ergibt -2.0
```

Anmerkung:

Der gebrochene Teil wird einfach abgeschnitten – es gibt also keine Rundung. Das Ergebnis ist von Typ Real (nicht Integer).

(6) Die SQR()-Funktion

Die Funktion berechnet die Quadratwurzel des Arguments.

Syntax:

```
SQR(<real expression>)
```

Im Fall eines negativen Arguments wird ein Fehlerzustand vom Typ Z (vgl. Fehler: Referenz nicht gefunden) erzeugt.

(7) Die SIN() -Funktion

Die Funktion berechnet den Sinus des im Bogenmaß (Radiant) gegebenen Arguments.

Syntax:

```
SIN(<real expression>)
```

(8) Die COS()-Funktion

Die Funktion berechnet den Kosinus des im Bogenmaß (Radiant) gegebenen Arguments.

Syntax:

```
COS(<real expression>)
```

(9) Die TAN()-Funktion

Die Funktion berechnet den Tangens des im Bogenmaß (Radiant) gegebenen Arguments.

Syntax:

```
TAN(<real expression>)
```

Anmerkung:

Obwohl die Tangens-Funktion für `pi/2 ± n*pi` nicht definiert ist, löst die Funktion für solche Argumente keinen Fehler aus, was wohl auf Ungenauigkeiten bei der Rundung auf die interne Binärdarstellung zurückzuführen ist. So ergibt beispielsweise der Aufruf von

```
print tan(pi/2), atn(tan(pi/2)), atn(tan(pi/2))*180/pi
```

die Ausgabe

```
2.7342611E+09   1.5707963       90
```

Interessanterweise führt der Aufruf von

```
print sin(pi/2), cos(pi*0.5), sin(pi/2)/cos(pi/2),
atn(sin(pi/2)/cos(pi/2))*180/pi
```

dagegen zu

```
1    7.3145904E-10   1.3671306E+09   90
```

Der Anwendungsprogrammierer sollte sich also selbst um das Abfangen solcher Situationen kümmern.

(10) Die ASN()-Funktion

Die Funktion berechnet den Arkussinus des gegebenen Arguments und liefert ein Ergebnis im Bogenmaß (Radiant) zwischen –pi/2 und pi/2..

Syntax:

```
ASN(<real expression>)
```

Der Wert des als Argument übergebenen Ausdrucks muss im Wertebereich von -1 bis 1 liegen, andernfalls wird ein Fehlerzustand vom Typ Z (vgl. Fehler: Referenz nicht gefunden) erzeugt.

(11) Die ACS()-Funktion

Die Funktion berechnet den Arkuskosinus des gegebenen Arguments und liefert ein Ergebnis im Bogenmaß (Radiant) zwischen 0 und pi..

Syntax:

```
ACS(<real expression>)
```

Der Wert des als Argument übergebenen Ausdrucks muss im Wertebereich von -1 bis 1 liegen, andernfalls wird ein Fehlerzustand vom Typ Z (vgl. Fehler: Referenz nicht gefunden) erzeugt.

(12) Die ATN()-Funktion

Die Funktion berechnet den Arkustangens des gegebenen Arguments und liefert ein Ergebnis im Bogenmaß (Radiant) zwischen –pi/2 und pi/2..

Syntax:

```
ATN(<real expression>)
```

(13) Die EXP()-Funktion

Die Funktion liefert den Wert der Expotentialfunktion e^x, wobei *x* der Wert des übergebenen Arguments ist.

Syntax:

```
EXP(<real expression>)
```

(14) Die LOG() function

Die Funktion berechnet den Natürlichen Logarithmus *ln(x)* des Arguments.

Syntax:

```
LOG(<real expression>)
```

Der als Argument verwendete Ausdruck muss einen (echt) positiven Wert ergeben, andernfalls wird ein Fehlerzustand vom Typ Z (vgl. Fehler: Referenz nicht gefunden) erzeugt

(15) Die LGT()-Funktion

Die Funktion berechnet den Zehnerlogarithmus *lg(x)* des Arguments.

Syntax:

```
LGT(<real expression>)
```

Der als Argument verwendete Ausdruck muss einen (echt) positiven Wert ergeben, andernfalls wird ein Fehlerzustand vom Typ Z (vgl. Fehler: Referenz nicht gefunden) erzeugt

(16) Die HSN()-Funktion

Die Funktion berechnet den Sinus Hyperbolicus des Arguments.

Syntax:

```
HSN(<real expression>)
```

(17) Die HCS()-Funktion

Die Funktion berechnet den Kosinus Hyperbolicus des Arguments.

Syntax:

```
HCS(<real expression>)
```

(18) Die HTN()-Funktion

Die Funktion berechnet den Tangens Hyperbolicus des Arguments.

Syntax:

```
HTN(<real expression>)
```

(19) Benutzerdefinierter FN_name()-Funktionsaufruf

Dieses Konstrukt ermöglicht es, eine mit der FUNCTION-Anweisung (siehe 2.8) definierte benutzerdefinierte Funktion aufzurufen.

Syntax:

```
FN_<function name>[(<argument> {, <argument> })]
```

Dazu wird das Präfix FN_ dem Namen der Funktion vorangestellt.

(20) Vordefinierte Real-Konstanten

```
PI      hat den Wert 3.14159265.

EE      hat den Wert 2.71828183, also die Basis des Natürlichen Logarithmus
```

Anmerkung:

Erstaunlicherweise lässt der Compiler eine Zuweisung an diese Konstanten (z.B. pi=3) ohne eine Fehlermeldung zu; im erzeugten Code bleibt dies allerdings ohne Wirkung, d.h. der Wert dieser Konstanten wird nicht tatsächlich geändert. Offensichtlich wird bei der Zuweisung eine gleichnamige Variable angelegt, auf die jedoch nicht lesend zugegriffen werden kann.

(21) Die FLOAT()-Funktion

Die Funktion konvertiert das Ergebnis des als Argument übergebenen Integer-Ausdrucks in einen Real-Wert.

Syntax:

```
FLOAT(<integer expression>)
```

(22) Die VAL()-Funktion

Die Funktion wandelt eine Zeichenkette in einen Real-Wert um..

Syntax:

```
VAL(<string>)
```

Ergänzung:

Die VAL-Funktion interpretiert Zahlen in „normaler" und Expotentialdarstellung

Ist die übergebene Zeichenkette nicht vollständig als Real-Zahl interpretierbar, so wird kein Fehler ausgelöst, sondern es werden – vom Anfang beginnend - so viele Zeichen interpretiert, wie interpretierbar sind. Beginnt die Zeichenkette dagegen mit einem für Zahlen ungültigen Zeichen, so wird ein Laufzeitfehler vom Typ # (vgl. Fehler: Referenz nicht gefunden) ausgelöst . Führende Leerzeichen werden ignoriert.

Beispiel:

```
print val("1e2"), val("-1"), val("1.8"), val(" 1,8")
```

ergibt

```
100            -1              1.8             1
```

2.10.3 Funktionen für Zeichenketten

Diese Funktionen sind (nur) in String-Ausdrücken verwendbar.

(1) Die CHR$()-Funktion

Die Funktion gibt Zeichen-Code des Wertes des als Argument übergebenen Integer-Ausdrucks zurück.

Syntax:

```
CHR$(<integer expression>)
```

Anmerkung:

Genau genommen ist das Ergebnis eine Zeichenkette, die aus höchstens (falls der Wert des Ausdrucks nicht Null ergibt) einem Zeichen besteht. Der Zeichencode dieses Zeichens ist einfach der Wert des Low-Bytes des Integer-Wertes.

(2) Die RIGHT$()-Funktion

Die Funktion liefert die rechte Teilzeichenkette eines Strings mit (höchstens) der angegebenen Länge.

Syntax:

```
RIGHT$(<string>, <integer expression>)
```

Beispiel:

```
a$=right$("abcdefg", 3)
```

Hier wird die Zeichenkette “efg” zurückgegeben.

Anmerkung:

Ist die angegebene Länge größer als die Länge der Quell-Zeichenkette, wird die gesamte Zeichenkette zurückgegeben. Eine Längenangabe kleiner 1 (einschließlich negativer Werte) ergibt eine leere Zeichenkette.

(3) Die MID$() -Funktion

Die Funktion liefert eineTeilzeichenkette eines Strings, beginnend beim festgelegten Index, mit (höchstens) der angegebenen Länge.

Syntax:

```
MID$(<string>, <integer expression1> [, <integer expression2>])
```

Wird das zweite Integer-Argument (die Längenangabe) weggelassen, gehören alle Zeichen ab dem angegebenen Index zum Ergebnis..

Beispiel:

```
a$=mid$("abcdefg", 3, 2)
```

Hier wird die Zeichenkette "cd" zurückgegeben.

Anmerkung:

Die Indexzählung für den Beginn der Teilzeichenkette beginnt mit 1 (nicht mit 0).

Eine Längenangabe kleiner 1 (einschließlich negativer Werte) oder ein ungültiger Startindex ergibt eine leere Zeichenkette.

(4) Die LEFT$()-Funktion

Die Funktion liefert die linke Teilzeichenkette eines Strings mit (höchstens) der angegebenen Länge.

Syntax:

```
LEFT$(<string>, <integer expression>)
```

Beispiel:

```
a$=left$("abcdefg", 3)
```

Hier wird die zeichenkette “abc” zurückgegeben.

Anmerkung:

Ist die angegebene Länge größer als die Länge der Quell-Zeichenkette, wird die gesamte Zeichenkette zurückgegeben. Eine Längenangabe kleiner 1 (einschließlich negativer Werte) ergibt eine leere Zeichenkette.

(5) Die STR$()-Funktion

Die Funktion wandelt das Ergebnis eines numerischen Ausdrucks in eine Zeichenkette um..

Syntax:

```
STR$([%<format>] <integer or real expression>)
```

Dabei kann eine Formatierung in der gleichen Weise wie für die PRINT-Anweisung (siehe 3) verwendet werden.

Beispiel:

```
STR$(%n.m X0)
```

(6) Die INKEY$-Funktion

Die Funktion gibt ein Zeichen Eingabezeichen als Zeichenkette zurück..

Syntax:

```
INKEY$
```

Beispiel:

```
10 if inkey$="" then 10
```

Hinweis: Mehrfachzuweisungen wie a$=a$+inkey$ lösen eine Garbage-Collection aus.

Ergänzung:

INKEY$ liest ein einzelnes Zeichen ohne Bildschirmecho von der Tastatur, wobei nicht auf eine Eingabe gewartet wird. Ist zum Zeitpunkt des Aufrufs keine Taste gedrückt (bzw. steht kein Zeichen im Tastaturpuffer, so ist das Ergebnis eine leere Zeichenkette.

Intern wird für INKEY$ die DOS-Funktion 06H (Direct console I/O) mit DL=FFH (Zeicheneingabe) verwendet.

3 Fortgeschrittene Programmierung

3.1 Compiler-Optionen

```
A>TBC /C file name<cr> bzw. A>TBC file name /C<cr>
```

Kommandozeilenoptionen werden wie im Beispiel oben, also am Anfang oder am Ende der Parameterliste beim Aufruf des Compilers, notiert..

Alternativ ist es möglich, die Optionen in der ersten Zeile des zu übersetzenden Quelltextes in der Form

```
options /c /a
```

zu notieren.

Folgende Optionen können angegeben werden:

Anmerkung:

Die Wirkung einiger der nachfolgenden Optionen ließ sich nicht genau bestimmen bzw. überprüfen. Für diese Optionen wurde lediglich die Beschreibung aus der Originaldokumentation übersetzt. Nicht weiter überprüfte Optionen sind mit (?) am Ende des Beschreibungstextes gekennzeichnet.

(1) Option /A

Die Option erzwingt die explizite Deklaration aller numerischen Variablen als `integer` oder `real`.

(2) Option /V

Die Option erzwingt die explizite Deklaration aller numerischen Variablen, deren Name 6 oder mehr Zeichen lang ist, als `integer` oder `real`.

(3) Option /S

Mit dieser Option erhalten nicht explizit deklarierte Variablen, deren Name nur aus einem einzigen Buchstaben besteht, automatisch den Typ `integer`.

(4) Option /F

Die Option legt die intern verwendeten Ausgaberoutinen für die Anweisungen PRINT#0 und LOCATE entsprechend nachfolgender Tabelle fest.

Option	Ausgaberoutine	Cursor-Steuerung
default	int 29h	ah=6, int 21h
/f0	int 29h	int 29h
/f1	ah=6, int 21h	ah=6, int 21h
/f2	ah=6, int 21h	ah=2, int 21h
/f3	ah=2, int 21h	ah=2, int 21h

(5) Option /W

Mit dieser Option lässt sich die Initialisierung der Pufferbereiche beeinflussen. Die in der nachfolgenden Übersicht angegebenen Werte sind die voreingestellten Standardwerte. (?)

```
/wt=130     temporary buffer size (read etc.)
/wi=128     input buffer size (max. 256)
/wb=256     string operation buffer size
/wl=&fff8   upper limit of string area (in data segment)
/ws=4096    stack area
/wp=&fffe   stack pointer
```

Ergänzung zur Speicheraufteilung:

Bei der Initialisierung wird das Datensegment offensichtlich auf dem nächsten freien Paragraphen oberhalb des Codesegments (dessen Größe sich aus dem Programmcode ergibt) angelegt.

Für die Initialisierung des Stack-Segments wird zum Wert von DS die Anzahl der entsprechend der Stackgröße (Option /ws) benötigte Anzahl von Paragraphen addiert.

Mit den Optionen /wl und /wp lassen sich zwar die nutzbaren Größen in Stack und Datensegment reduzieren. Einen praktischen Nutzen hat dies jedoch nicht, weil der Bereich für erweiterte Arrays (mit DIMX definiert) immer 64K oberhalb des Stack-Segments beginnt. Das bedeutet, dass die Segmentadresse des ersten erweiterten Feldes stets SS+1000H ist. Außerdem verlangen mit TOKIWA-Basic übersetzte Programme aus diesem Grund, dass oberhalb des Stacksegment-Wertes mindestens 64K freier Speicher vorhanden sein müssen.

Erweiterte Felder beginnen stets an Paragraphengrenzen (durch 16 teilbare Adressen). Die Adressen selbst werden im Datensegment fortlaufend abgelegt. Das Anlegen der Felder erfolgt während der Programminitialisierung.

Oberhalb dieser Segment-Adressliste befinden sich vermutlich die festen Puffer, gefolgt von der Liste der globalen Variablen. Diese Liste enthält für numerische Variablen die Variablenwerte selbst, für Strings deren Adressen (Offsets in DS). Die Elemente einfacher Arrays (mit DIM definiert) liegen ebenfalls fortlaufend in diesem Bereich.

Achtung: Die Verringerung des Wertes der Option /wl verkleinert den nutzbaren Bereich im Datensegment, ohne dass die Einhaltung dieser Restriktion vom Compiler geprüft wird. Dies kann dann zu Laufzeitfehlern beim Zugriff auf Variablen führen. Der Wert von /wp darf (auch das überprüft der Compiler nicht) nicht kleiner als der von /wl sein, da sich sonst Stack und Datenbereiche wechselseitig überschreiben können.

(6) Option /J

Mit dieser Option wird die Generierung von JMP-Befehlen zum Überspringen von mit SUBROUTINE oder FUNCTION deklarierten benutzerdefinierten Unterprogrammen unterdrückt.

Ergänzung:

Benutzerdefinierte Unterprogramme und Funktionen können im Prinzip irgendwo innerhalb des Quelltextes stehen und werden vom Compiler nicht selbstständig an ausgezeichneten Stellen zusammengefasst. Deshalb wird normalerweise vor jeder dieser Routinen eine Sprunganweisung hinter ihr Ende eingefügt, um den Ablauf des eigentlichen Programms nicht zu stören.

Fasst man alle benutzerdefinierten Routinen am Anfang oder Ende des Quelltextes zusammen, lässt sich mit der Option /J die Größe des erzeugten Programms etwas reduzieren. In diesem Fall ist der Programmierer dann selbst dafür zuständig den entsprechenden Bereich mit einer GOTO-Anweisung zu überspringen.

(7) Option /U

Mit dieser Option wird der Stringpuffer nach der Verarbeitung von Zeichenketten erneuert. (?)

(8) Option /E

Die Option gibt Fehlermeldungen in eine Datei (BC.ERR) aus.

Ergänzung:

Die Datei BC.ERR wird im selben Verzeichnis wie die Quelldatei angelegt und beinhaltet eine Kopie vom Compiler sonst nur auf die Konsole ausgegebenen Fehlermeldungen während des Übersetzungsvorganges. Die Option hat keinen Einfluss auf die Ausgabe der Meldungen von Laufzeitfehlern – sie landen immer auf der Konsole.

(9) Option /T

Die Option erlaubt die Ablaufverfolgung (Trace) des kompilierten Objekts. (?)

(10) Option /TL

Die Option listet den Programmablauf.

Ergänzung:

Mit dieser Option werden während des Programmablaufs einige Zeilennummern aus dem Quelltext, jeweils in eckigen Klammern, zusätzlich zu den normalen Programmausgaben auf die Konsole geschrieben. Soweit sich feststellen ließ, werden jeweils die Zeilennummer vor einer Verzweigung (Sprung, Unterprogrammaufruf …), die Zeilennummer der Verzweigung selbst sowie die Zeile nach der Rückkehr aus einem Unterprogramm ausgegeben.

(11) Option /C

Mit dieser Option wird eine .COM-Datei statt einer .EXE-Datei als Ausgabe erzeugt.

Anmerkung:

Weil bei COM-Dateien der EXE-Header entfällt, ist hier das erzeugte Programm etwas kleiner.

Beide Varianten belegen beim Programmstart komplett den größten verfügbaren Speicherblock in DOS (also nahezu den ganzen verfügbaren Speicher), so dass das Starten von Kindprozessen oder die Anforderung von Speicher über die DOS-Funktion 48H ohne größere Tricks nicht möglich ist.

(12) Option /O

Die Option schaltet die Optimierung ein.

Anmerkung:

Soweit sich feststellen ließ, bedeutet Optimierung die Ausrichtung aller Symbole (Variablen, Einsprungadressen für Prozeduren) an geradzahligen Adressen. Daraus folgt eine etwas größere Ausgabedatei. Auf Systemen mit 16-Bit-Datenbus (also nicht beim i8088) bewirkt die Ausrichtung zumeist eine etwas höhere Ausführungsgeschwindigkeit.

(13) Option /#x

Die Übersetzung erfolgt mit den Bedingungen, die am Zeilenanfang definiert sind. So verwendet man #e bzw. #c um .EXE- bzw. .COM-Dateien zu erzeugen.

Ergänzung:

Die Beschreibung ist etwas missverständlich. Etwas ausführlicher lässt sich folgendes sagen:

Tokiwa Basic unterstützt eine einfach Form bedingter Übersetzung. Mit der Option /#, gefolgt von einem Buchstaben, lässt sich dieser Buchstabe gewissermaßen als Symbol für die bedingte Übersetzung definieren. Leitet man eine Programmzeile mit # und nachfolgendem Buchstaben ein, so wird sie nur dann übersetzt, wenn der betreffende Buchstabe mit der Option /#x als Symbol definiert wurde.

Es gibt darüber hinaus die vordefinierten Symbole c (beim Erzeugen einer COM-Datei) und e (beim Erzeugen einer EXE-Datei), in Abhängigkeit von der Option /c. Für ihren Einsatz ist keine Angabe der /#x-Option erforderlich.

Beispiel:

```
options /#y /c
#y print "Symbol y defined"
#c print "Program is in COM format"
#e print "Program is in EXE format"
```

Eine praktische Anwendung ist dem ENVIRON.BAS zu entnehmen, wo in Abhängigkeit davon, ob das Programm als EXE- oder COM-Datei erstellt wird, das Bestimmen der PSP-Adresse auf unterschiedliche Weise erfolgt..

(14) Option /I

Die Option legt fest, dass in Unterprogrammen deklarierte Variablen statisch sind.

Ergänzung:

Genau genommen betrifft diese Option nur lokale Variablen vom Integer-Typ, da andere lokale Variablen ohnehin immer statisch sind (vgl. Ergänzung zu 1).

Ist die Option gesetzt, werden die innerhalb eines benannten Unterprogramms bzw. einer Funktion definierten lokalen Integer-Variablen nicht auf dem Stack, sondern global im Datensegment angelegt. Ihre Sichtbarkeit bleibt dennoch lokal.

(15) Option /Q

Mit dieser Option wird beim Auftreten eines Syntaxfehlers auf einen Tastendruck gewartet.

3.2 Trace-Modus

Bei eingeschaltetem Trace-Modus kann die Ablaufverfolgung mit folgenden Tastenkombinationen interaktiv gesteuert werden:

```
Key code      Aktion

 ^Z            Ablauf unterbrechen (Programm anhalten)
 ^L            Listing des Ablaufs einschalten
 ^Q            Listing des Ablaufs ausschalten
 ^C            Rückkehr zum System (Programmabbruch)
```

Beispiel: `A>TBC /TL HANOI<cr>`

Innerhalb des Quelltextes kann die Ablaufverfolgung abschnittsweise zu- bzw. abgeschaltet werden.

Syntax:

```
TRACE on
TRACE off
```

3.3 Referenzierung von Zeilennummern

Mit einem At-Symbol (@) eingeleitete Label-Bezeichner, Namen von Unterprogrammen und Zeilennummern können zur Referenzierung einer Zeile verwendet werden.

3.4 Arbeiten mit der VALUE-Anweisung

(1) Einfache Integer-Verarbeitung

Beispiel:

```
value a+b
```

Das Ergebnis verbleibt im AX-Register; es erfolgt keine Zuweisung an eine Variable

(2) Fortsetzung eines Ausdrucks

Syntax:

```
VALUE ! <operator> <subsequent integer expression>
```

Das Ausrufezeichen (!) repräsentiert dabei den aktuellen Wert des AX-Registers.

Beispiel:

```
value !+3
```

Hier wird 3 zum aktuellen Wert des AX-Registers addiert.

(3) Zuweisung

Syntax:

```
VALUE = <integer or string variable>
```

Der Inhalt des AX-Registers wird der Variablen zugewiesen

3.5 Beispiele zur Assembler-Programmierung

(1) Referenzieren von BASIC-Variablen

```
integer var, ary, exary
dim ary(10)
dimx exary(20)

/mov    ax,var
/mov    var,ax
/mov    ax,ary(2)
/mov    es,exary()
/inc    var
/call   var
/jmp    var
```

Dabei dürfen die Register (ax, cx, dx, al, cl, dl, etc.) sowie reservierte Wörter des Assemblers (siehe 2.9.9) nicht als Variablen verwendet werden.

Zuweisungen von/an 8-Bit-Register(n):

```
/mov    al,var
/mov    ary(2),bl
/inc    byte ptr var
```

Offset-Adressen:

```
/mov    bx,offset var
/mov    si,offset ary(2)
```

```
/mov    ax,[bx+si+offset ary(0)]
```

Zugriff auf Parameter in SUBROUTINE- und FUNCTION-Anweisungen:

```
/mov    ax,arg
/inc    arg
```

Dies wird übersetzt `in  mov ax,[bp+n]` und inc `word ptr [bp+n]`.

Ergänzung:

Unterprogramme und Funktionen werden in TOKIWA-Basic nach C-Konventionen aufgerufen. Dies bedeutet, dass die Argumente (die hier grundsätzlich 16 Bit breit sind) in umgekehrter Reihenfolge, das heißt das letzte zuerst, auf den Stack gelegt werden. Zuletzt wird vor dem Aufruf die Rückkehradresse (16-Bit-Offset im Codesegment – es gibt nur NEAR-Aufrufe) auf den Stack gelegt. Das Abräumen des Stacks übernimmt der Aufrufer.

Lokale (Integer-)Variablen stehen in der Reihenfolge ihrer Deklaration „unterhalb“ der Rückkehradresse. Sie werden unmittelbar beim Eintritt in ein Unterprogramm auf den Stack gelegt.
Das Unterprogramm muss den Wert des BP-Registers sichern und den Stack-Pointer (SP) bei der Rückkehr auf den Wert von BP setzen.

Beim Eintritt in ein Unterprogramm sieht der Stackframe demnach wie folgt aus:

```
BP+2n       arg n
…           …
BP+4        arg2
BP+2        arg1
BP          return offset
BP-2        local integer variable 1
BP-2        local integer variable 2
…           …
SP=BP-2m    local integer variable m
```

Lokale Real- und String-Variablen sind statisch. Das bedeutet, sie sind zwar nur lokal sichtbar, werden aber nicht auf den Stack, sondern global im Datensegment angelegt.

(2) Referenzieren von Zeilennummern und Labels

Als Offset-Adressen:

```
/mov    bx,@100
/mov    [bx+si+@100],cx
/mov    ax,[@100+2]
/inc    byte ptr [@100]
```

Als Sprungziele

```
/jmp    @100
/call   @100+2
/jz     @name
```

Ergänzung:

Das @-Symbol liefert im Inline-Assembler die Offset-Adresse eines Symbols im Codesegment. Es ist auf Zeilennummern ebenso anwendbar, wie auf Label-Bezeichner oder die Bezeichner

benannter Unterprogramme bzw. Funktionen, wobei der Offset hier gleichbedeutend mit der Einspriungadresse ist.

(3) BYTE-Definition

```
/byte 5, 'A', "ABCDE", @100, @TEST
```

Hierbei werden `@100` und `@TEST` in relative Adressen umgewandelt.

(4) WORD-Definition

```
/word 1000, @200, @TEST
```

Hierbei werden @200 und @test in absolute Adressen innerhalb des Codesegments umgewandelt.

(5) BLANK-Definition

```
/blank  10
```

Mit dieser Anweisung werden 10 Bytes innerhalb des Codesegments reserviert.

(6) Sonstiges

Definition von Segmenten als Präfix

```
/cs
/mov    [di], ax
```

oder

```
/cs  /mov    ax,[di]
```

Definition von Anweisungswiederholungen:

```
/rep
/stosb
```

oder

```
/rep  /movsb
/rep  movsw
```

Offset-Definitionen als Index:

```
/mov    ax,[bx+(@100+2)*3/2+'AB'+_const]
/mov    al,[bx+si]+@100/16+5
/mov    bx,offset ary(0)+3
```

Dabei sind nur einfache Operationen erlaubt.

3.6 Assembler und die PRINT-Anweisung

Ergebnisse, die in Assembler-Abschnitten berechnet wurden, können leicht mit Hilfe des !-Symbols (das den Inhalt des AX-Registers repräsentiert) an die PRINT-Anweisung übergeben werden.

Beispiel:

```
/mov    ax,7
/shl    ax,1
print !
```

Das Ergebnis sollte 14 sein – 7, um ein Bit nach links verschoben.

Beispiel zur Ausgabe der Adresse des Datensegments:

```
/mov    ax,ds
print %4h !
```

3.7 Beispielprogramme

(1) Beiliegende Programmquellen

LNEQ.BAS dient der Lösung linearer Gleichungssysteme. Verwenden Sie zunächst die Beispieldaten in der Datei "MATRIX.DAT". Probieren Sie dann folgende Gleichungen aus:

```
 x + 2y = 5
3x + 4y = 7
```

Die Daten für diese Gleichungen sind 2,1,2,5,3,4,7, wobei die erste Zahl die Anzahl der unabhängigen Gleichungen angibt.

HANOI.BAS das Problem der Türme von Hanoi. Dies ist ein Beispiel für rekursive Aufrufe – ähnlich denen in PASCAL.

ENVIRON.BAS ist ein Beispiel für den Zugriff auf Kommandozeilen-Parameter und Environment-Variablen..

TCALC.BAS schließlich ist ein Beispiel für einen Rechner nach der Methode der Umgekehrten Polnischen Notation.

(2) Ein ausführbares Grafikprogramm (PC/AT und Kompatible)

```
MANDELB.EXE
```

Dieses Programm zeichnet die Mandelbrot-Menge auf den Bildschirm. Um es übersetzen zu können, ist die p_set()-Funktion aus der Bibliothek VGAGRAPH.LIB erforderlich.

```
merge "vgagraph.lib"
defint i-n
r1 = -2.
r2 = .5
h1 = -1.25
h2 = 1.25
p = (r2 - r1) / 399.
q = (h2 - h1) / 399.

for k = 0 to 399
  c = k * p + r1
  n = 120 + k
  for j = 0 to 399
    d = j * q + h1
    y = d
    x = c
    for i = 0 to 40
      xx = x * x
      yy = y * y
      xy = x * y
      y = xy + xy + d
      x = xx + c - yy
      if xx + yy > 4. then call p_set( n, j, i mod 7 + 1): break
    next i
  next j
next k
```

3.8 Fehlermeldungen des Compilers

```
Array over
Endif error
Illegal end

Label over
Miss match
Nest error
Nest over
No File
Phase error
Prog over

Step error
Sub or Fnend error
Swend error
Sub & Fn over
SW ?
Syntax error
Var over
WR error
#Array
#Var
#Label
#Sub & Fn

#               double declaration of variable.
$               unassigned string variable.
%               format error.
)               missing parenthesis.
```

```
-               no corresponding to FOR, WHILE, REPEAT.
?               no label to refer.
A               undefined array.
I               undefined or unassigned integer variable.
J               ranges over for relative jump.
O               no OPEN# statement
R               unassigned real variable.
S               undefined subroutine or function
U               undefined variables at /A, /V option.
V               disagreement with NEXT variable.
```

3.9 Laufzeit-Fehlermeldungen

```
#               numerical data
$               string data
/               delimiter
D               out of data
E               end of file
F               no file
I               integer
M               range over at string area
O               overflow
Z               fatal
No file
error
```

3.10 Zum Schluss

Dieses Handbuch wurde aus dem Japanischen (ins Englische) übersetzt. Ich danke Ihnen, dass Sie bereit sind, ein so unvollkommenes Handbuch dennoch zu lesen.

December 30, 2000

by Dr. Genji OKADA
E-Mail: okada@en3.ehime-u.ac.jp

Die Deutsche Übersetzung geht etwas über die ursprüngliche Fassung hinaus, in dem der Versuch unternommen wurde, offene Fragen des Originals zu klären, um so die Verwendung des Compilers zu erleichtern. Daraus ergeben sich eine Reihe von Anmerkungen und Ergänzungen, die im Text durch Einrahmung der jeweiligen Abschnitte gekennzeichnet sind.

31 Dezember 2012

R.-Erik Ebert
E-Mail: ebert@kiezsoft.de