robotron

Programmtechnische
Beschreibung

TBASIC

C3315-0400-1 M3030

DCP 3.30

Programmtechnische Beschreibung

TBASIC

VEB Robotron Buchungsmaschinenwerk
Karl-Marx-Stadt
Postschließfach 129
Karl-Marx-Stadt
9 0 1 0

C3315-0400-1 M3030

Die vorliegende 1. Auflage der Dokumentation "Bedienungsanleitung und Sprachbeschreibung fuer TBASIC" unter DCP 3.30 entspricht dem Stand vom 01.09.1988 und unterliegt nicht dem Aenderungsdienst.

Die Dokumentation wurde durch ein Kollektiv des

VEB Robotron Buchungsmaschinenwerk Karl - Marx -Stadt

erarbeitet.

Bitte senden Sie uns Ihre Hinweise, Kritiken, Wuensche oder Forderungen zur Dokumentation zu.

Herausgeber:

VEB Robotron-Buchungsmaschinenwerk
Karl-Marx-Stadt
VEB Robotron-Bueromaschinenwerk "Ernst Thaelmann"
Soemmerda

Ag706/ 42/90 III/9/1

I N H A L T S V E R Z E I C H N I S

1. Einleitung

TBASIC ist eine eigenstaendige Programmierumgebung fuer den PC EC 1834 und besteht aus einem Editor, einem Compiler, einer Laufzeitbibliothek und einem eingebauten Linker, der die einzelnen Programmteile miteinander verbindet.
TBASIC ist aufwaertskompatibel mit dem BASIC-Interpreter BASI.

In BASIC geschriebene Programme sind aufgrund der erweiterten Strukturierungsmoeglichkeiten wesentlich uebersichtlicher und einfacher zu korrigieren.

Definierbare Funktionen und Prozeduren ermoeglichen die Deklaration lokaler, statischer und globaler Variablen sowie echte Rekursion.

Es wird ein neuer Zugriffsmodus ("BINARY") bereitgestellt, bei dem eine Datei als Folge von einzelnen Bytes behandelt werden kann.

Maschinencode laesst sich direkt in den Programmcode einfuegen, damit wird das Einbinden ganzer Objektcode-Dateien ermoeglicht.

Fuer TBASIC ist die gesamte Sprachbeschreibung des BASIC-Interpreters BASI gueltig. In dieser vorliegenden Dokumentation werden nur die Abweichungen und Zusaetze zur BASI-Sprachbeschreibung und die Bedienung von TBASIC beschrieben.

2. TBASIC - Benutzung und Menue

2.1. Benutzung von TBASIC

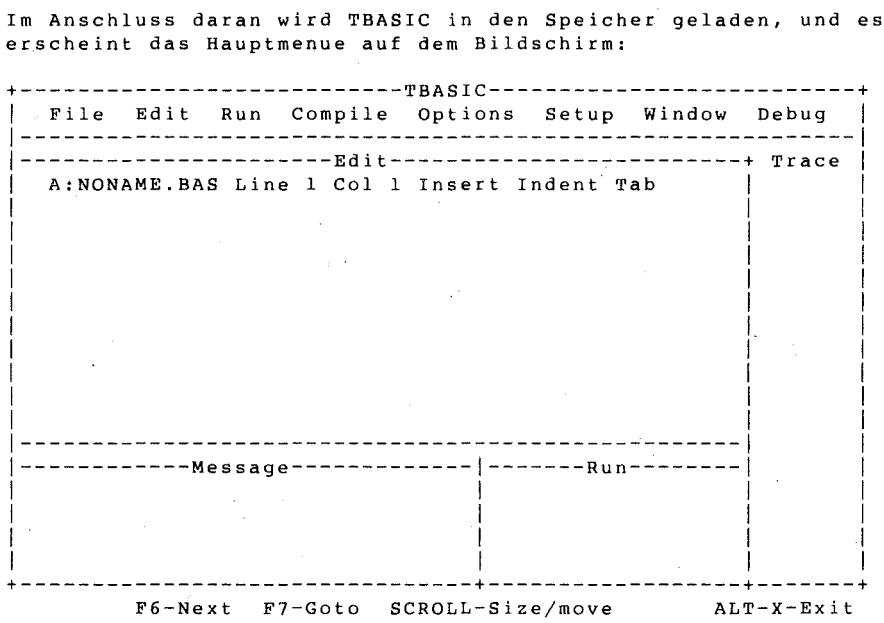
Folgende Dateien gehoeren zu TBASIC:

TBASIC.EXE TBASIC-Programmpaket.
Beinhaltet den Editor, Compiler und die Laufzeitbibliothek.

Der Aufruf von TBASIC erfolgt durch Eingabe von TBASIC und Druecken der Taste <ENTER>:

A> TBASIC <ENTER>

Im Anschluss daran wird TBASIC in den Speicher geladen, und es erscheint das Hauptmenue auf dem Bildschirm:



Am oberen Rand des Bildschirms werden die zur Verfuegung stehenden Kommandos angezeigt. Darunter befinden sich die Fenster **Edit** zur Texteingabe und -bearbeitung, **Message** zur Anzeige von Meldungen des Compilers und des Systems, **Trace** zur Fehlersuche und **Run** zur Ausgabe eines laufenden Programms.

Die Auswahl der Kommandos im Hauptmenue kann erfolgen durch:

- Eingabe des Anfangsbuchstabens ueber die Tastatur

- Bewegen des hervorgehobenen dargestellten Teils der Liste mit Hilfe der Kursorsteuertasten auf das gewuenschte Kommando und Druecken der Taste <ENTER>.

Die Rueckkehr von einem Untermenue in das Hauptmenue erfolgt durch Druecken der Taste <ESC>.

2.2. Aufbau von TBASIC

2.2.1. Fenster

Die Programmierumgebung von TBASIC arbeitet mit vier verschiedenen Fenstern:

Edit Eingabe und Aendern von Quelltexten

Trace Ausgabe der Zeilennummern eines laufenden Programms

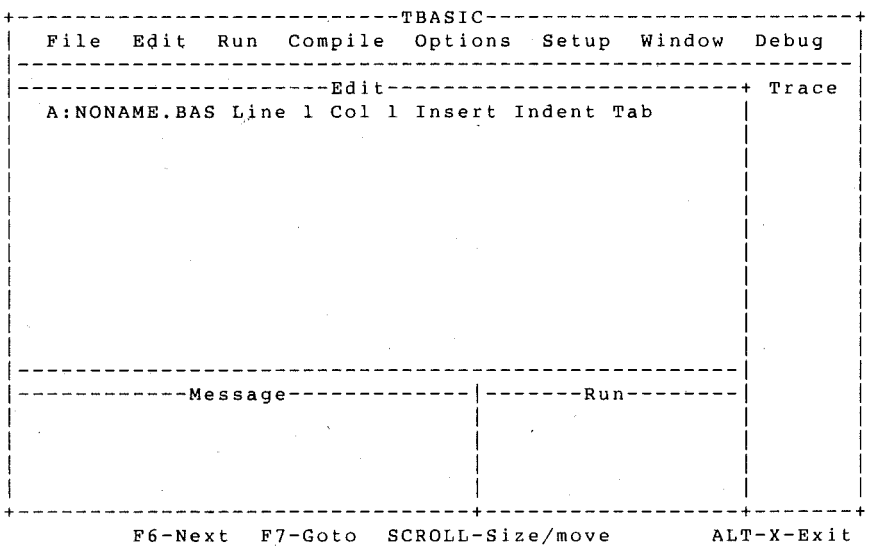
Message Ausgabe von Meldungen, z.B. Ergebnisse des Compilers

Run Anzeige der Ausgaben eines Anwenderprogrammes

Ein Fenster kann mit dem Kommando **Open** aus dem Menue **Windows** eroeffnet werden.
Nach Druecken der Taste <SROLL LOCK> kann das aktive Fenster mit Hilfe der Kursorsteuertasten an jede beliebige Stelle des Bildschirms bewegt werden.
Nach nochmaligem Druecken der Taste <SROLL LOCK> erhalten die Kursortasten ihre normale Funktion zurueck.
Mit Hilfe der Funktionstaste F5 (ZOOM) kann die Groesse der Fenster **Edit** und **Run** geaendert werden (umschalten auf gesamten Bildschirm und zurueck).

2.2.2. Editor

Der in TBASIC integrierte Editor wurde speziell auf die Erstellung von Quelltexten ausgelegt.
Durch Eingabe des Kommandos **E** bzw. Bewegen des hervorgehobenen Teils des Balkens auf das Fenster **Edit** und Druecken von <ENTER> im Hauptmenue wird der Editor aktiviert. Das Fenster **Edit** rueckt in den Vordergrund, der Editor erwartet Eingaben und Kommandos.



Die oberste Zeile ist die **Statuszeile**. Hierbei haben die einzelnen Angaben folgende Bedeutung:

Line Anzeige der aktuellen Nummer der Zeile, in der sich der Kursor befindet.
Die erste Zeile eines Textes hat die Nummer 1, die zweite die Nummer 2, usw.

Col Anzeige der aktuellen Spaltenposition des Kursors.

Insert Einfuegemodus eingeschaltet.
Neu eingegebene Zeichen werden ab der Position des Kursors in den Text eingefuegt.
Mit den Tastenbefehlen <INS> oder ^V (^ = Taste <CTRL>) kann zwischen Einfuegen ("Insert") und Ueberschreiben ("Overwrite") umgeschaltet werden. Bei Ueberschreiben wird der alte Text ab Kursorposition durch die neue Eingabe ueberschrieben.

Indent Anzeige, ob automatische Tabulierfunktion eingeschaltet ist.
An- und Abschalten erfolgt durch ^OJ.

Tab Taste <TAB> bewegt den Kursor zu Tabulatorstopps.
Ab- und Ausschalten erfolgt durch das Kommando ^OT.

x:Dateiname.typ

Gibt das Laufwerk, den Namen und den Typ der Datei an, die augenblicklich vom Editor bearbeitet wird.
Wurde noch kein Name angegeben, so wird der Standardname **NONAME.BAS** verwendet.

Der Editor realisiert verschiedene Kommandos zur Bewegung des Kursors, dem Suchen und Ersetzen von Textteilen usw., die in vier Gruppen eingeteilt werden koennen:

- Bewegen des Kursors
- Einfuegen und Loeschen
- Operationen mit Textbloecken
- Verschiedenes

Uebersicht der Editorkommandos	Tasten
Bewegen des Kursors	
Zeichen nach links	CTRL-S oder <--
Zeichen nach rechts	CTRL-D oder -->
Wort nach links	CTRL-A
Wort nach rechts	CTRL-F
Zeile nach oben	CTRL-E oder ↑
Zeile nach unten	CTRL-X oder ↓
Aufwaerts rollen	CTRL-W
Abwaerts rollen	CTRL-Z
Seite nach oben	CTRL-R oder PgUp
Seite nach unten	CTRL-C oder PgDn
Zeile links	CTRL-Q-S oder HOME
Zeile rechts	CTRL-Q-D oder END
Oberer Bildschirmrand	CTRL-Q-E
Unterer Bildschirmrand	CTRL-Q-X
Textbeginn	CTRL-Q-R oder CTRL-PgUp
Textende	CTRL-Q-C oder CTRL-PgDn
Blockanfang	CTRL-Q-B
Blockende	CTRL-Q-K
Letzte Kursorposition	CTRL-Q-P
Einfuegen und Loeschen	
Einfuegemodus ein/aus	CTRL-V oder INS
Zeile einfuegen	CTRL-N
Zeile loeschen	CTRL-Y
Loeschen bis Zeilenende	CTRL-Q-Y
Linkes Zeichen loeschen	CTRL-H oder BACKSPACE
Zeichen an Kursorposition loeschen	CTRL-G oder DEL
Rechtes Wort loeschen	CTRL-T

Blockoperationen

Blockbeginn markieren	CTRL-K-B oder F7
Blockende markieren	CTRL-K-K oder F8
Einzelnes Wort als Block	CTRL-K-T
Block kopieren	CTRL-K-C
Block verschieben	CTRL-K-V
Block loeschen	CTRL-K-Y
Block von Diskette lesen	CTRL-K-R
Block auf Diskette schreiben	CTRL-K-W
Block verdecken/anzeigen	CTRL-K-H

Verschiedenes

Editor ohne Speichern beenden	CTRL-K-D, CTRL-K-Q oder ESC
Editor mit Speichern beenden	CTRL-K-S oder F2
Loeschen, neue Datei anlegen	F3
Tabulator	CTRL-I oder TAB
Tab-Modus an/aus	CTRL-O-T
Autom. Tabulieren ein/aus	CTRL-O-I
Ruecknahme von Aenderungen	CTRL-Q-L
Marke setzen	CTRL-Kn
Zu einer Marke springen	CTRL-Qn
Suchen	CTRL-Q-F
Suchen und Ersetzen	CTRL-Q-A
Suchen/Ersetzen wiederholen	CTRL-L
Einleitung fuer Steuerzeichen	CTRL-P
Operation abbrechen	CTRL-U
Fehlermeldung zurueckholen	CTRL-Q-W

Bewegen des Kursors

Zeichen nach links **CTRL-S**

Bewegt den Kursor um ein Zeichen nach links, ohne dieses Zeichen zu veraendern. Ist der linke Rand erreicht, stoppt der Kursor.

Zeichen nach rechts **CTRL-D**

Bewegt den Kursor um ein Zeichen nach rechts, ohne dieses Zeichen zu veraendern. Erreicht der Kursor den rechten Rand des Bildschirms, wird der gesamte Inhalt des Bildschirms um eine Spalte nach links verschoben. Maximal sind 254 Spalten moeglich, danach stoppt der Kursor.

Wort nach links **CTRL-A**

Setzt den Kursor auf den Anfang des Wortes, das links von der momentanen Kursorposition steht. Die folgenden Zeichen werden vom Editor als Trennzeichen zwischen zwei Worten verstanden: das Leerzeichen und < > , : . ( ) [ ] ^ ' = + - / $. Befindet sich in der momentanen Zeile kein weiterer Wortanfang, wird zur vorhergehenden Zeile gesprungen.

Wort nach rechts **CTRL-F**
Setzt den Kursor auf den Anfang des Wortes, das rechts von der momentanen Kursorposition steht. Es gelten dieselben Bedingungen wie bei "Wort links". Befindet sich in der momentanen Zeile kein weiteres Wort, wird zur naechsten Zeile gesprungen.

Zeile nach oben **CTRL-E oder ↑**
Bewegt den Kursor um eine Zeile nach oben. Ist die oberste Zeile im Fenster erreicht, rollt der Fensterinhalt um eine Zeile nach unten.

Zeile nach unten **CTRL-X oder ↓**
Bewegt den Kursor um eine Zeile nach unten. Ist die unterste Zeile im Fenster erreicht, rollt der Fensterinhalt um eine Zeile nach oben.

Aufwaerts rollen **CTRL-W**
Rollt den Fensterinhalt in Richtung zum Textanfang. Der gesamte Inhalt des Fensters wird um eine Zeile nach unten verschoben. Die Position des Kursors innerhalb der Datei bleibt solange unveraendert, bis er die unterste Fensterzeile erreicht.

Abwaerts rollen **CTRL-Z**
Rollt den Fensterinhalt in Richtung zum Textende. Der gesamte Inhalt des Fensters wird um eine Zeile nach oben verschoben. Die Position des Kursors innerhalb der Datei bleibt solange unveraendert, bis er die oberste Fensterzeile erreicht.

Seite nach oben **CTRL-R oder PgUp**
Bewegt den Kursor um eine Bildschirmseite in Richtung auf den Textanfang. Die Textzeile, die zuvor als oberste im Textfenster sichtbar war, steht nun als unterste im Bild.

Seite nach unten **CTRL-C oder PgDn**
Bewegt den Kursor um eine Bildschirmseite in Richtung auf das Textende. Die Textzeile, die zuvor als unterste im Textfenster sichtbar war, steht nun als oberste im Bild.

Zeile links **CTRL-Q-S oder HOME**
Bewegt den Kursor auf den aeusseren linken Rand des Fensters, d.h. in die Textspalte 1.

Zeile rechts **CTRL-Q-D oder END**
Bewegt den Kursor auf das rechte Ende der momentanen Zeile, d.h. auf die Position des letzten Zeichens (Leerzeichen werden nicht beruecksichtigt).
Ist die Zeile breiter als das Fenster, wird entsprechend horizontal gerollt.

Oberer Bildschirmrand **CTRL-Q-E**
Setzt den Kursor auf die oberste Zeile des Fensters, ohne dabei die Spaltenposition zu veraendern.

Unterer Bildschirmrand **CTRL-Q-X**
Setzt den Kursor auf die unterste Zeile des Fensters, ohne dabei die Spaltenposition zu veraendern.

Textbeginn **CTRL-Q-R oder CTRL-PgUp**
Setzt den Kursor auf das erste Zeichen des Textes.

Textende **CTRL-Q-C oder CTRL-PgDn**
Setzt den Kursor auf das letzte Zeichen des Textes.

Mit einer Kombination aus CTRL-Q und den Buchstaben B,K und L laesst sich der Kursor zu bestimmten Stellen innerhalb des Textes bewegen:

Blockanfang **CTRL-Q-B**
Setzt den Kursor auf die Position, die zuvor mit CTRL-K-B als Blockbeginn markiert wurde. Dieses Kommando funktioniert auch dann, wenn Blockmarkierungen mit "Verdecken" unsichtbar gemacht worden sind. Wurde kein Block markiert, springt CTRL-Q-B zum Anfang des Textes.

Blockende **CTRL-Q-K**
Setzt den Kursor auf die Position, die zuvor mit CTRL-K-K als Blockende markiert wurde. Dieses Kommando funktioniert auch dann, wenn Blockmarkierungen mit "Verdecken" unsichtbar gemacht worden sind. Wurde kein Block markiert, springt CTRL-Q-K zum Anfang des Textes.

Letzte Kursorposition **CTRL-Q-P**
Setzt den Kursor auf die Position, die er vor der Ausfuehrung des letzten Kommandos hatte und laesst sich beispielsweise nach "Suchen" oder "Speichern" gut verwenden.

Einfuegen und Loeschen

Mit den in diesem Abschnitt beschriebenen Kommandos koennen Zeichen, Woerter und Zeilen eingefuegt ("Insert") oder geloescht ("Delete") werden.

Einfuegen an/aus **CTRL-V oder INS**
Bei der Eingabe von Text koennen zwei Grundmodi -**Einfuegen und Ueberschreiben**- gewaehlt werden, zwischen denen mit diesem Kommando gewechselt werden kann. Die Voreinstellung beim ersten Start des Editors ist **Einfuegen:** Bei der Eingabe von Zeichen werden rechts vom Kursor stehende Textteile entsprechend verschoben. Wenn sich der Editor dagegen im Modus **Ueberschreiben** befindet, dann ersetzen neue Eingaben rechts vom Kursor stehenden Text zeichenweise, der dabei ueberschriebene Text geht verloren. Ein Druck auf <ENTER> fuegt in diesem Modus keine neue Zeile ein, sondern bewegt lediglich den Kursor. Der momentan aktive Modus wird in der Statuszeile des Editors angezeigt.

Linkes Zeichen loeschen **CTRL-H oder BACKSPACE**
Bewegt den Kursor um eine Spalte nach links und loescht das dort befindliche Zeichen. Rechts vom Kursor stehender Text wird automatisch um eine Spalte nach links gerueckt. Mit diesem Kommando koennen auch Zeilenvorschuebe geloescht werden.

Zeichen an Kursorposition loeschen **CTRL-G oder DEL**
Loescht das Zeichen, auf dem der Kursor steht, und verschiebt alle rechts von der Kursorposition befindlichen Zeichen um eine Spalte nach links.

Rechtes Wort loeschen **CTRL-T**
Loescht das Wort rechts von der momentanen Kursorposition. Unter einem "Wort" versteht der Editor eine beliebig lange Folge von Zeichen, die mit einem der folgenden Zeichen endet: dem Leerzeichen sowie < > , : . ( ) [ ] ^ ' " + - / und $. Befindet sich innerhalb der momentanen Zeile rechts vom Kursor kein weiteres Wort mehr, dann loescht CTRL-T den naechsten Zeilenvorschub und das erste "Wort" bzw. fuehrende Leerzeichen der naechsten Zeile.

Zeile einfuegen **CTRL-N**
Fuegt einen Zeilenvorschub an der Position des Kursors ein, ohne den Kursor dabei zu bewegen. Die Eingabe von <ENTER> setzt den Kursor dagegen auf die Spalte 1 der neu eingefuegten Zeile.

Zeile loeschen **CTRL-Y**
Loescht die Zeile, auf der sich der Kursor momentan befindet, und bewegt alle folgenden Zeilen um eine Zeile nach oben. Der Kursor wird in die erste Spalte der naechsten Zeile gesetzt. Dieses Kommando kann mit CTRL-Q-L nicht rueckgaengig gemacht werden!

Loeschen bis Zeilenende **CTRL-Q-Y**
Loescht alle Zeichen bis zum Ende der momentanen Zeile, beginnend mit dem Zeichen, auf dem sich der Kursor momentan befindet. Im Gegensatz zu "Zeile loeschen" kann dieses Kommando mit CTRL-Q-L wieder rueckgaengig gemacht werden, solange die Zeile noch nicht verlassen ist.

Block-Operationen

Unter einem Block versteht der Editor eine (fast) beliebige Textmenge - angefangen von einem einzigen Zeichen bis hin zu mehreren hundert Zeilen - die von zwei speziellen Markierungen eingeschlossen ist. Der Anfang des Blocks wird mit dem Kommando "Blockanfang", sein Ende mit dem Kommando "Blockende" markiert. Es kann immer nur ein Block gesetzt und bearbeitet werden.

Ein markierter Block kann innerhalb des Quelltextes kopiert, verschoben oder geloescht werden. Es ist auch moeglich, ihn als eigenstaendige Datei auf die Diskette zu schreiben. Ausserdem kennt der Editor ein Kommando, mit dem eine Textdatei von der Diskette als "Block" in den momentan bearbeitenden Text eingelesen werden kann.

Blockbeginn markieren **CTRL-K-B oder F7**
Haelt die momentane Position des Kursors als Markierung fuer den Blockbeginn fest. Die Markierung selbst ist auf dem Bildschirm nicht sichtbar, der Inhalt des Blocks wird erst dann hervorgehoben dargestellt, wenn auch eine Blockende-Markierung gesetzt ist, die sich hinter dem Blockbeginn befinden muss. Der Editor setzt beim Start oder nach dem Einlesen einer neuen Datei automatisch einen Blockbeginn am Anfang des Textes.

Blockende markieren **CTRL-K-K oder F8**
Haelt die momentane Position des Kursors als Markierung fuer das Blockende fest. Die Markierung selbst ist auf dem Bildschirm nicht sichtbar, der Inhalt des Blocks wird erst dann hervorgehoben dargestellt, wenn auch eine Blockbeginn-Markierung gesetzt ist, die sich vor dem Blockende befinden muss.

Einzelnes Wort als Block **CTRL-K-T**
Dieses Kommando setzt Blockbeginn und -ende automatisch um ein einzelnes Wort herum. Wenn sich der Kursor inmitten eines Wortes befindet, wird dieses Wort als Block markiert. Befindet sich der Kursor zwischen zwei Worten, dann wird das linke der beiden Woerter als Block gekennzeichnet. Die Definition von "Wort" ist hier dieselbe wie bei dem Kommando "Wort loeschen" und "Wort links/Wort rechts".

Block verdecken/zeigen **CTRL-K-H**
Schaltet bei der Darstellung eines markierten Blocks zwischen "hervorgehoben" und "normal" hin und her. Spruenge zum Blockbeginn und -ende arbeiten unabhaengig von der Darstellungsart- die Blockbearbeitungskommandos (Kopieren, Verschieben, Loeschen und Schreiben) dagegen nur, wenn der Block hervorgehoben dargestellt, also nicht "verdeckt" ist.

Block kopieren **CTRL-K-C**
Kopiert einen zuvor markierten Block so, dass sein erstes Zeichen auf der momentanen Position des Kursors zu stehen kommt. Um beispielsweise einen Block an das Ende des Quelltextes zu kopieren, muss zuerst der Kursor an das Textende bewegt und danach dieses Kommando gegeben werden. Der Originalblock bleibt unveraendert, die Markierungen fuer Blockbeginn und -ende werden auf den neu kopierten Block gesetzt. Ist kein Block markiert oder befindet sich der Kursor zum Zeitpunkt des Kommandos innerhalb des markierten Blockes, wird das Kommando nicht ausgefuehrt.

Block verschieben **CTRL-K-V**

Verschiebt einen zuvor markierten Block so, dass sein erstes Zeichen auf der momentanen Position des Kursors zu stehen kommt (vgl. "Block kopieren"). Der Original-Block wird geloescht, die Markierungen fuer Blockbeginn und -ende werden auf den neu verschobenen Block gesetzt. Ist kein Block markiert oder befindet sich der Kursor zum Zeitpunkt des Kommandos innerhalb des markierten Blocks, wird das Kommando nicht ausgefuehrt.

Block loeschen **CTRL-K-Y**

Loescht einen zuvor markierten Block vollstaendig aus dem Speicher, die Position des Kursors spielt dabei keine Rolle. **Achtung!** Dieses Kommando ist nicht rueckgaengig zu machen!

Block auf Diskette schreiben **CTRL-K-W**

Speichert einen zuvor markierten Block als eigenstaendige Textdatei auf der Diskette ab. Der Original-Block im Speicher bleibt unveraendert, die Markierungen fuer Blockbeginn und -ende ebenfalls. Wird dieses Kommando gegeben, fordert der Editor die Eingabe eines Namens. Wird kein Dateityp mit angegeben, haengt TBASIC automatisch den Typ ".BAS" an. Existiert die bezeichnete Datei bereits, wird eine entsprechende Meldung ausgegeben, und es entsteht die Moeglichkeit zum Abbruch der Operation bzw. der Eingabe eines neuen Dateinamens. Ist zum Zeitpunkt des Kommandos kein Block markiert, wird das Kommando nicht ausgefuehrt.

Block von Diskette lesen **CTRL-K-R**

Dieses Kommando behandelt eine Diskettendatei als "Block". Der Name der Datei muss angegeben werden, wobei die gleichen Regeln wie fuer "Block schreiben" gelten. Die bezeichnete Datei wird komplett in den momentan bearbeiteten Text eingelesen, die Diskettendatei selbst bleibt unveraendert. Fuer das Einlesen gelten die gleichen Regeln wie fuer "Block kopieren": Das erste Zeichen kommt auf der momentanen Position des Kursors zu stehen. Hinter dem zukuenftigen Block stehender Text wird entspechend verschoben. Die Markierungen fuer Blockbeginn und -ende sind nach Ausfuehrung der Operation auf den eingelesenen Text gesetzt.

Editor

F7 ≙ CTRL-K-B Beginn Block

F8 ≙ CTRL-K-K Ende Block

Verschiedenes

Dieser Abschnitt behandelt die restlichen Kommandos des Editors, die in keine der bereits behandelten Kategorien fallen.

Editor ohne Speichern beenden **CTRL-K-D, CTRL-K-Q oder ESC**
Beendet den Editor und kehrt zum Hauptmenue von TBASIC zurueck, ohne den Quelltext zu speichern. Eine Speicherung des Textes kann entweder mit **File Save**, durch Setzen des Schalters **Auto Save** im Untermenue **Miscellaneous** von **Setup** oder durch ein Kommando innerhalb des Editors erreicht werden.

Speichern im Editor **CTRL-K-S oder F2**
Speichert den momentan bearbeiteten Text, ohne dabei den Editor zu beenden. Der Kursor wird danach auf den Textanfang gesetzt - mit CTRL-Q-L kann auf die vorherige Position positioniert werden.

Neue Datei **F3**
Befindet sich zum Zeitpunkt dieses Kommandos entweder kein Text innerhalb des Editors oder dieser Text wurde seit der letzten Speicherung nicht veraendert, dann wird er ohne Rueckfrage geloescht. Ansonsten wird eine entsprechende Meldung ausgegeben, und es besteht die Moeglichkeit, die Operation abzubrechen. Nach dem Loeschen fragt TBASIC nach dem Namen der neuen Arbeitsdatei.

Tab **CTRL-I oder TAB**
Setzt den Kursor auf die naechste Tabulatorposition. Diese Positionen sind festgelegt und haben jeweils einen Abstand von acht Zeichen zueinander (Spalte 1, 9,17 usw.).

Tabulatoren an/aus **CTRL-O-T**
Schaltet die Eingabemoeglichkeit von Tabulatoren (siehe oben) an bzw. aus. Wird in der Statuszeile des Editors **Tab** angezeigt, koennen Bewegungen zum naechsten Tabulatorstopp von Hand eingegeben werden, das automatische Tabulieren des Editors ist ausgeschaltet. Ansonsten uebernimmt TBASIC das Einruecken selbst (siehe unten).

Automatische Tabulierung ein/aus **CTRL-O-I**
Ist die automatische Tabulierung ausgeschaltet, setzt die Eingabe eines Zeilenvorschubs (d.h. ein Druck auf <ENTER>) den Kursor auf die Spalte 1 der naechsten Zeile, durch die Eingabe von Tabulatoren (siehe oben) kann der Beginn der naechsten Eingabe nach rechts verschoben werden.

Ist die automatische Tabulierung dagegen an, dann merkt sich der Editor, an welcher Stelle der vorhergehenden Zeile mit der Eingabe von Text begonnen wurde und setzt den Kursor auch in der naechsten Zeile wieder in die gleiche Spalte. Soll eine folgende Zeile weiter links beginnen, muessen die Kursortasten oder <BACKSPACE> benutzt werden.

Diese Automatik ist zur Unterstuetzung von optisch strukturierten Listings gedacht - mit CTRL-O-I laesst sie sich ein- bzw. ausschalten. Im eingeschalteten Zustand wird das Wort **Indent** in der Statuszeile des Editors ausgegeben, im ausgeschalteten Zustand fehlt es.

Aenderungen zuruecknehmen **CTRL-Q-L**
Der Editor speichert eine Zeile erst dann endgueltig, wenn <ENTER> gedrueckt oder der Kursor in eine andere Zeile bewegt wird. Solange das nicht passiert ist, koennen mit diesem Kommando saemtliche Aenderungen innerhalb einer Zeile wieder rueckgaengig gemacht werden. (Das Loeschen mit CTRL-Y "verlaesst" die geloeschte Zeile sofort - deshalb ist hier ein Wiederherstellen nicht mehr moeglich.)

Markierung setzen **CTRL-Kn**
Es koennen bis zu vier verschiedene Marken innerhalb eines Textes erzeugt werden, indem der Kursor auf die entsprechende Stelle gesetzt wird und dann das Kommando CTRL-K, gefolgt von einer Ziffer (Ø bis 3) gegeben wird. Eine so gesetzte Marke kann sehr einfach mit dem Kommando CTRL-Qn wiedergefunden werden. (Eine typische Anwendung in einem langen Quelltext: Markieren der momentanen Position mit CTRL-KØ, "Blaettern" im Quelltext nach einem bereits geschriebenen Programmteil, und danach mit CTRL-QØ zurueck zur vorherigen Position).

Markierung finden **CTRL-Qn**
Dieses Kommando bewegt den Kursor zu einer von maximal vier zuvor gesetzten Markierungen. **n** steht fuer eine Ziffer im Bereich von Ø bis 3. Ist die entsprechende Markierung nicht gesetzt, wird das Kommando nicht ausgefuehrt.

Suchen **CTRL-Q-F**
Mit diesem Kommando kann eine Folge von maximal 30 Zeichen innerhalb des Textes gesucht werden. Der Editor reagiert mit dem Loeschen der Statuszeile und der Aufforderung, den Suchbegriff einzugeben. Der Suchbegriff kann aus einer beliebigen Folge von Zeichen (inklusive Leer- und Steuerzeichen) bestehen. Seine Eingabe wird mit <ENTER> abgeschlossen. Steuerzeichen innerhalb des Suchbegriffs muessen mit dem Kommando CTRL-P eingeleitet werden: Um beispielsweise einen Zeilenvorschub zu suchen, muss die Tastenfolge CTRL-P CTRL-N CTRL-P CTRL-J benutzt werden.

Waehrend der Eingabe stehen die Korrekturmoeglichkeiten "Zeichen links/rechts" und "Wort links/rechts" zur Verfuegung. "Wort rechts" stellt dabei ein zuvor geloeschtes Zeichen bzw. ein Zeichen der letzten Eingabe wieder her. "Wort rechts" macht das gleiche fuer die gesamte Eingabe. Das Kommando "Suchen" kann zu diesem Zeitpunkt durch die Eingabe von CTRL-U abgebrochen werden.

Nach Angabe des Suchbegriffs fragt der Editor nach Optionen fuer "Suchen". Dabei stehen die folgenden Moeglichkeiten zur Verfuegung:

B Suchen rueckwaerts ("backwards") von der momentanen Position des Kursors aus in Richtung Textbeginn.

G Suchen innerhalb des gesamten Textes ("global"), unabhaengig von der momentanen Position des Kursors.

n Suchen nach dem n-ten Vorkommen. n steht fuer eine ganze Zahl im Bereich von 1 bis 65535.

U Ignorieren von Gross- und Kleinschreibung ("uppercase")

W Suchen nach ganzen Worten: Ein Suchbegriff wie "aus" wird ignoriert, wenn er innerhalb des Wortes "Maus" steht.

Suchen und Ersetzen **CTRL-Q-A**

Dieses Kommando arbeitet in der gleichen Weise wie "Suchen". Nach der Eingabe des Suchbegriffs wird zusaetzlich nach einer Zeichenfolge gefragt, mit der eventuelle Fundstellen ersetzt werden sollen. Dieser Ersatz darf ebenfalls nur 30 Zeichen umfassen. Waehrend der Eingabe stehen die gleichen Moeglichkeiten wie bei "Suchen" zur Verfuegung. Die Laengen von Such- und Ersatzbegriff muessen nicht uebereinstimmen. Wird beispielsweise nur <ENTER> als Ersatzbegriff eingegeben, so wird der gefundene Begriff geloescht.

Die Optionen sind die gleichen wie bei "Suchen", hier kommt allerdings noch eine weitere Moeglichkeit dazu:

N Ersetzen ohne Rueckfrage. Wird diese Option nicht angegeben, fragt der Editor bei jeder Fundstelle explizit zurueck, ob sie ersetzt werden soll oder nicht.

Das Druecken von <ENTER> bei der Eingabe der Optionen (soweit vorhanden) startet den Suchvorgang. Wird der Suchbegriff gefunden und ist nicht die Option N angegeben, wird der Kursor auf das letzte Zeichen der Fundstelle gesetzt. Gleichzeitig erscheint in der Statuszeile des Editors die Anfrage **Replace (Y/N)?** (= Ersetzen (Ja/Nein). An diesem Punkt kann die Anfrage entweder beantwortet oder die Operation durch die Eingabe von CTRL-U abgebrochen werden.

Der jeweils letzte Aufruf von "Suchen und Ersetzen" kann durch die Eingabe von CTRL-L beliebig oft wiederholt werden.

Wiederholung Suchen/Ersetzen **CTRL-L**

Mit diesem Kommando kann der jeweils letzte Such- bzw. Such-/Austauschbefehl so wiederholt werden, als waeren die dazugehoerigen Informationen erneut eingegeben worden.

Einleitung fuer Steuerzeichen **CTRL-P**

Der Editor erlaubt die Eingabe von Steuerzeichen nicht nur in einem Such- oder Austauschbegriff, sondern auch innerhalb des Quelltextes. (Moeglichst nur fuer Kommantare verwenden.) Um ein Steuerzeichen einzugeben, muss zuerst CTRL-P und danach das Steuerzeichen selbst eingegeben werden. Steuerzeichen innerhalb des Quelltextes werden mit einem besonderen Zeichenattribut dargestellt.

Unterbrechung **CTRL-U**

Mit diesem Kommando kann jede Operation des Editors abgebrochen werden, wenn diese Operation eine Eingabe erwartet, also bei Rueckfragen von "Suchen/Ersetzen" sowie "Block lesen/-schreiben".

Fehlermeldung zurueckholen **CTRL-Q-W**

Wenn der Compiler einen Fehler innerhalb des Quelltextes entdeckt, aktiviert er den Editor und belegt dessen Statuszeile mit dem Text der Fehlermeldung. Normalerweise verschwindet diese Meldung nach dem ersten Tastendruck. Mit diesem Kommando kann sie erneut sichtbar gemacht werden.

2.2.3. Compiler

Der Aufruf des Compilers erfolgt vom Hauptmenue aus durch Eingabe von **C**. Ist ein "Main File" gesetzt, so erfolgt die Anfrage, ob die momentane "Arbeitsdatei" gespeichert werden soll. Danach erfolgt das Laden des "Main Files" und die Uebersetzung.
Ist kein "Main File" gesetzt, wird der Text compiliert, der sich im Speicher des Editors befindet.
Das uebersetzte Maschinenprogramm wird entweder im Hauptspeicher abgelegt (Standard) oder auf der Diskette aufgezeichnet (abhaengig vom Schalter **Compile to** im Menue **Options**).

Waehrend der Uebersetzung wird die Nummer jeder zwanzigsten Zeile im Fenster **Message** angezeigt.
Wird ein Fehler erkannt, aktiviert der Compiler automatisch den Editor und setzt den Kursor an die entsprechende Stelle. In der Statuszeile des Editors erscheint die Fehlermeldung. Diese verlischt nach dem ersten Tastendruck, kann aber jederzeit mit CTRL-Q-W wiederhergestellt werden.
Nach Abschluss einer fehlerfreien Compilierung enthaelt das Fenster **Message** einige zusaetzliche Informationen, wie z.B. Anzahl der uebersetzten Zeilen, Dauer des Compilierens, Laenge des Objektcodes sowie Groesse des Stack und des Datenbereiches in hexadezimaler Form.

Arbeitsschritte zum Erstellen eines Programms

1. Im Hauptmenue Auswahl des Untermenues **Files** und dort **New**. Auf die Anfrage "Work File" muss der Name des Programms eingegeben und anschliessend die Taste <ENTER> gedrueckt werden.

2. Aktivieren des Editors durch Eingabe von **E** und Eingabe des Programms (ohne Zeilennummern).
   Verlassen des Editors durch Druecken der Taste <ESC> und damit Rueckkehr zum Hauptmenue.

3. Aufruf des Compilers durch Eingabe von **C** und damit Uebersetzen des Programms, das sich momentan als "Work File" im Speicher befindet. Werden beim Uebersetzen keine syntaktischen Fehler festgestellt, so wird automatisch der Editor aktiviert.
   Die oberste Zeile des Fensters **Edit** enthaelt eine Meldung ueber die Art des Fehlers. Der Kursor steht an der Stelle im Programm, an der der Compiler den Fehler bemerkt hat. Der Fehler ist zu korrigieren. Anschliessend wird der Editor durch Druecken von <ESC> verlassen und der Compiler durch Eingabe von C erneut aufgerufen.

4. Start des Programms mit dem Kommando R.
   Die Ausgaben des Programms erscheinen im Fenster **Run**. Tritt bei der Abarbeitung ein Fehler auf, so geht TBASIC automatisch in den Editor ueber. Der Kursor wird auf die Stelle des Quelltextes gesetzt, an der der Fehler aufgetreten ist.

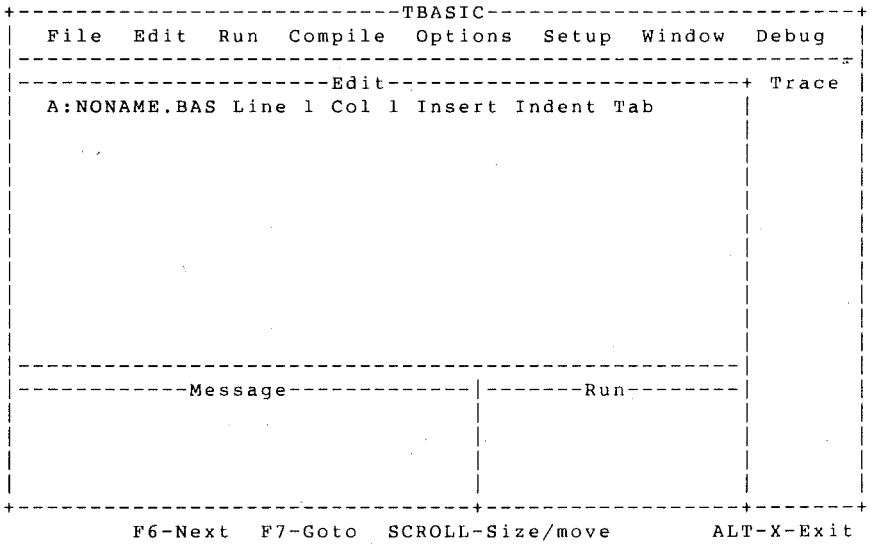
5. Speichern des Quelltextes erfolgt durch Aufruf des Untermenues **Files** und dort **Save**.
   Um das compilierte Programm (.EXE) abspeichern zu koennen, muss im Hauptmenue das Untermenue **Options** ausgewaehlt und dort **Compile to** auf **Executable File** umgeschaltet werden.
   Mit <ESC> kann dieses Untermenue verlassen werden.
   Ein anschliessendes Compilieren schreibt das ausfuehrbare Programm auf Diskette.

2.3. Hauptmenue von TBASIC

Die Bedienung von TBASIC setzt sich aus einer Folge von Tastendruckbefehlen und der Auswahl entsprechender Optionen aus mehreren Menue-Ebenen zusammen.
Nachfolgend werden die einzelnen Menues und die Arbeit mit den vier Fenstern der Programmierumgebung beschrieben.

TBASIC kann aus jedem Menue heraus mit **<ALT-X>** beendet werden.

Befindet sich zu diesem Zeitpunkt ein veraenderter (d.h. noch nicht gespeicherter) Text (Programm) im Speicher des Editors, gibt TBASIC eine entsprechende Meldung aus, und es besteht die Moeglichkeit der Speicherung.



Im Menuesystem stehen folgende Tastenkommandos zur Verfuegung:

Taste	**Funktion**
F2	Speichern der momentanen Arbeitsdatei
F3	Erzeugen einer neuen Arbeitsdatei
F5	Vergroessern/Verkleinern der Fenster Edit/ Run
F6	Setzen des jeweils naechsten Fensters aktiv
F7	Setzen des Kursors in das aktive Fenster
SROLL LOCK	Bewegen des aktiven Fensters mittels Kursorsteuertasten
ESC	Zurueck zum Hauptmenue (bzw. in die naechsthoehere Menue-Ebene)
ALT-C	Aufruf des Compilers
ALT-R	Starten des Programms
ALT-X	Beenden TBASIC

Im Editor sind weitere Tastenkommandos moeglich:

F2	Speichern der momentanen Arbeitsdatei
F3	Laden einer neuen Arbeitsdatei

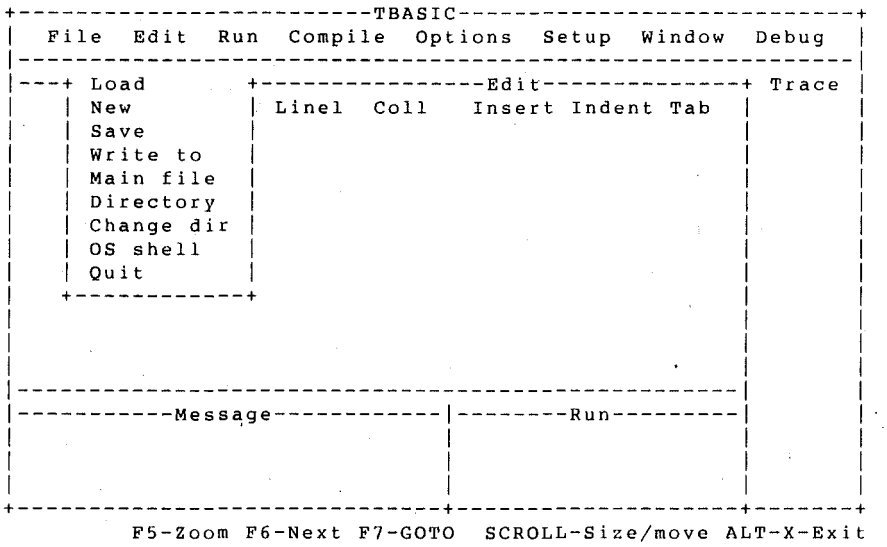
Um innerhalb des Hauptmenues eine Auswahl zu treffen, kann entweder der entsprechende Buchstabe eingegeben werden, oder der hervorgehoben dargestellte Teil des Balkens wird mit den Kursorsteuertasten an die entsprechende Stelle bewegt und anschliessend die Taste <ENTER> betaetigt. Mit Druecken der Taste <ESC> erfolgt die Rueckkehr in das naechsthoehere Menue.

Aus dem Fenster **Edit** oder aus einem Menue heraus kann jederzeit ein Kommando des Hauptmenues aufgerufen werden, indem die Taste <ALT> gedrueckt und der entsprechende Buchstabe eingegeben werden.
Das Festlegen von Bildschirmfarben, Suchpfaden und Fenstergroessen geschieht ueber das Menue **Setup**.

Alle gesetzten Optionen koennen in einer Datei vom Typ .TB einzeln bzw. zusammen mit TBASIC.EXE gespeichert werden.
TBASIC benutzt diese Optionen dann bei jedem Start automatisch. Von TBASIC wird als Standarddateiname zur Speicherung dieser Optionen der Name TBCONFIG.TB vorgeschlagen. Es kann aber auch jeder andere Dateiname gewaehlt werden.

2.3.1. Menue Files

Auf die Eingabe des Buchstabens **F** reagiert TBASIC mit der Ausgabe des Menues **Files**, das folgendermassen aufgebaut ist:



Load
Mit diesem Kommando kann eine Textdatei in den Speicher des Editors geladen werden. Nach Betaetigung des Buchstabens L (bzw. dem Bewegen des hervorgehoben dargestellten Balkens mit den Kursorsteuertasten und einem Druck auf <ENTER>) fragt TBASIC nach einem Dateinamen und gibt als Standard .BAS vor.

Zwei verschiedene Arten von Eingaben sind moeglich:

1. Ein Dateiname, der den Regeln von DCP entspricht. Wird kein Dateityp angegeben, dann wird automatisch der Dateityp .BAS hinzugefuegt. Um eine Datei zu laden, die keinen Dateityp besitzt, muss als letztes Zeichen der Eingabe ein Punkt folgen.

2. Ein Dateiname aus einem Directory. Wird nur die <ENTER>-Taste gedrueckt oder ein Dateiname angegeben, der einen Stern oder Fragezeichen enthaelt, dann reagiert TBASIC mit der Ausgabe des Inhaltsverzeichnisses des momentan aktiven Directory (bzw. des Standardlaufwerkes). Mit den Kursorsteuertasten, HOME, END, PgUp und PgDn kann der Kursor innerhalb der Dateiliste bewegt werden. Ein Druck auf <ENTER> waehlt eine Datei aus. Wird anstelle von <ENTER> ein <ESC> eingegeben, wird die Liste vom Bildschirm geloescht, und es erscheint eine erneute Anfrage nach dem Dateinamen.

Zu beachten ist, dass bestimmte Dateitypen (z.B.: .EXE, .COM, .BAT) nicht verwendet werden sollten, da sie fuer DCP reserviert sind.

Nach Eingabe des Dateinamens durchsucht TBASIC das momentan aktive Directory (bzw. Standardlaufwerk) nach der angegebenen Datei. Wird sie nicht gefunden, erscheint eine Fehlermeldung. Zum Anlegen einer neuen Datei muss das Kommando **NEW** benutzt werden.

Save
Schreibt die Arbeitsdatei (den Inhalt des Editor-Textspeichers) auf die Diskette. Falls bereits eine Datei gleichen Namens existiert (d.h. bereits eine aeltere Version des Quelltextes auf der Diskette gespeichert ist), dann erhaelt diese Datei den Dateityp .BAK. Traegt die Arbeitsdatei den Namen NONAME.BAS (noch kein Name angegeben), dann erfolgt die Aufforderung zur Eingabe eines Dateinamens. Wird anstelle einer Eingabe die Taste <ENTER> gedrueckt, wird der Text unter dem Namen NONAME.BAS gespeichert.

Auch hier kann durch die Eingabe von Stern und Fragezeichen innerhalb des Dateinamens ein Inhaltsverzeichnis ausgegeben und der Dateiname aus diesem Verzeichnis mit den Kursorsteuertasten, HOME, END, PgUp und PgDn ausgewaehlt werden. <ESC> anstelle einer Auswahl bringt die Anzeige **File Name** zurueck.

Main file
Bei der Arbeit mit sehr grossen Programmen empfiehlt es sich, den Quelltext in mehrere Abschnitte zu unterteilen. Die einzelnen Teile eines derartigen Quelltextes werden als eigenstaendige "Include"-Dateien gespeichert und bearbeitet - nur bei der Compilierung gilt der gesamte Text als zusammenhaengendes Stueck.

TBASIC unterscheidet dabei zwischen zwei Arten von Dateien: der Hauptdatei ("Main File") und den einzelnen Arbeitsdateien ("Work File"). Die Hauptdatei enthaelt an einer oder mehreren Stellen den Compiler-Befehl **$INCLUDE**, gefolgt von dem Namen einer Arbeitsdatei. Bei der Uebersetzung liest der Compiler die Hauptdatei und fuegt die einzelnen Arbeitsdateien jeweils an den entsprechenden Stellen ein.

Der Compiler-Befehl **$INCLUDE** ist im Kapitel 4 ausfuehrlich beschrieben.

Um ein "Main File" mit dem Editor zu bearbeiten, wird sein Name einfach als "Work File" angegeben, darauf wird diese Datei in den Editor geladen.
Wird zu einem spaeteren Zeitpunkt ein anderes "Work File" angegeben, wechselt TBASIC lediglich die Arbeitsdatei, die Angabe des "Main File" bleibt erhalten.

Beim Start des Compilers wird geprueft, ob sich die Namen von "Main File" und dem momentanen "Work File" unterscheiden. Ist das nicht der Fall (oder ist ueberhaupt kein "Main File" gesetzt), dann uebersetzt der Compiler die momentan im Speicher befindliche Arbeitsdatei. Unterscheiden sich die beiden Namen, dann fragt TBASIC zurueck, wenn die Arbeitsdatei seit dem letzten Speichern veraendert wurde, und ermoeglicht das Speichern, bevor das "Main File" geladen wird. Wird waehrend des Compilierens ein Fehler gefunden, dann laedt der Compiler automatisch die Datei in den Speicher des Editors, in der der Fehler aufgetreten ist, und setzt ihren Namen als "Work File". Nach der Korrektur des Fehlers und einem erneuten Aufruf des Compilers wiederholt sich der Ablauf.

New
Loescht den Arbeitsspeicher des Editors. Falls sich hier noch nicht abgespeicherter Text befinden sollte, erscheint eine entsprechende Meldung auf dem Bildschirm und die Moeglichkeit zum Speichern.

Write to
Mit dem Kommando kann der Quelltext unter einem neuen Namen gespeichert werden. Bei der Anfrage nach dem neuen Namen kann (wie bei Save und Load) durch die Eingabe von Stern oder Fragezeichen ein Inhaltsverzeichnis ausgegeben und der neue Name aus dieser Liste ausgewaehlt oder der Name direkt eingegeben werden.

Directory
Gibt das Inhaltsverzeichnis des momentan gesetzten Directories aus und kann (durch die Eingabe des entsprechenden Suchwegs) auch zur Anzeige anderer Directories benutzt werden.

Change dir
Setzt ein anderes Directory als Standard-Directory. Es muss der Suchpfad eingegeben werden, der den Regeln von DCP entsprechen muss. Der Abschnitt ueber das Menue **Setup** enthaelt noch einige wichtige Details ueber die Verwendung von Directories durch TBASIC selbst.

OS shell
Kehrt zur Kommandoebene von DCP zurueck, wobei TBASIC im Hauptspeicher verbleibt. Danach koennen beliebige DCP-Kommandos ausgefuehrt werden. Die Eingabe des Befehls EXIT aktiviert wieder TBASIC.

Quit
Beendet TBASIC und kehrt zur Kommandoebene von DCP zurueck. Falls sich zu diesem Zeitpunkt noch ein nicht abgespeicherter Quelltext im Editor befindet, erfolgt eine entsprechende Rueckfrage und die Moeglichkeit zum Speichern. Ein Druck auf <ALT-X> in einer beliebigen Menue-Ebene von TBASIC hat die gleiche Wirkung.

2.3.2. Kommando Edit

Mit diesem Kommando wird der bildschirmorientierte Editor aktiviert. Mit ihm kann der Inhalt der Datei bearbeitet werden, die zuvor geladen wurde. Falls sich zum Zeitpunkt des Kommandos keine Arbeitsdatei im Speicher befindet, gibt TBASIC den Namen NONAME.BAS vor.

2.3.3. Kommando Run

Mit diesem Kommando wird ein uebersetztes Programm gestartet, das sich im Speicher des Computers befindet.

Ein sofortiger Start ist nur dann moeglich, wenn:

- sich ein uebersetztes Programm im Speicher befindet.
- der Quelltext dieses Programms seit dem letzten Compilieren nicht veraendert wurde.
- seit dem letzten Aufruf des Compilers keine Compiler-Optionen veraendert wurden.

Ansonsten wird zuerst der Quelltext durch einen Aufruf des Compilers neu uebersetzt. Befindet sich kein Quelltext im Speicher, dann erfolgt eine Anfrage, und TBASIC laedt eine entsprechende Datei vor dem Uebersetzen.

Das Abarbeiten des Programms kann mit <CTRL-BREAK> abgebrochen werden. Dies ist nur dann moeglich, wenn die Option **Keyboard break** gesetzt ist und das Programm Ein- oder Ausgaben beinhaltet. Tritt waehrend der Programmausfuehrung ein Fehler auf, dann wird der Editor aktiviert. Der Kursor steht auf der entsprechenden Stelle im Quelltext, die Statuszeile des Editors enthaelt die Fehlermeldung. Die Fehlermeldung verschwindet automatisch nach dem ersten Tastendruck zur Korrektur des Fehlers.

Ausgaben des Programms werden im Fenster **Run** dargestellt. Im Fenster **Message** zaehlt TBASIC die Anzahl der ausgefuehrten Befehle und Programmzeilen in Schritten zu je 20 mit.

Mit <ALT-F5> kann das Fenster **Run** waehrend der Programmausfuehrung zwischen der vorgegebenen Groesse und dem gesamten Bildschirm umgeschaltet werden.

2.3.4. Kommando Compile

Ein Druck auf die Taste **C** oder ein Positionieren des hervorgehoben dargestellten Teils des Balkens auf **Compile** und ein Druck auf <ENTER> innerhalb des Hauptmenues ruft den Compiler auf. Er uebersetzt den im Speicher des Editors befindlichen Quelltext in einzelne Maschinenbefehle. Ist ein "Main File" gesetzt und befindet sich diese Datei momentan nicht im Speicher, wird sie zuerst geladen (siehe Menue **Files**, Abschnitt **Main File**). Ist weder eine Haupt- noch eine Arbeitsdatei angegeben, fordert TBASIC die Eingabe eines Dateinamens. Diese Datei wird dann in den Speicher des Editors geladen und uebersetzt.

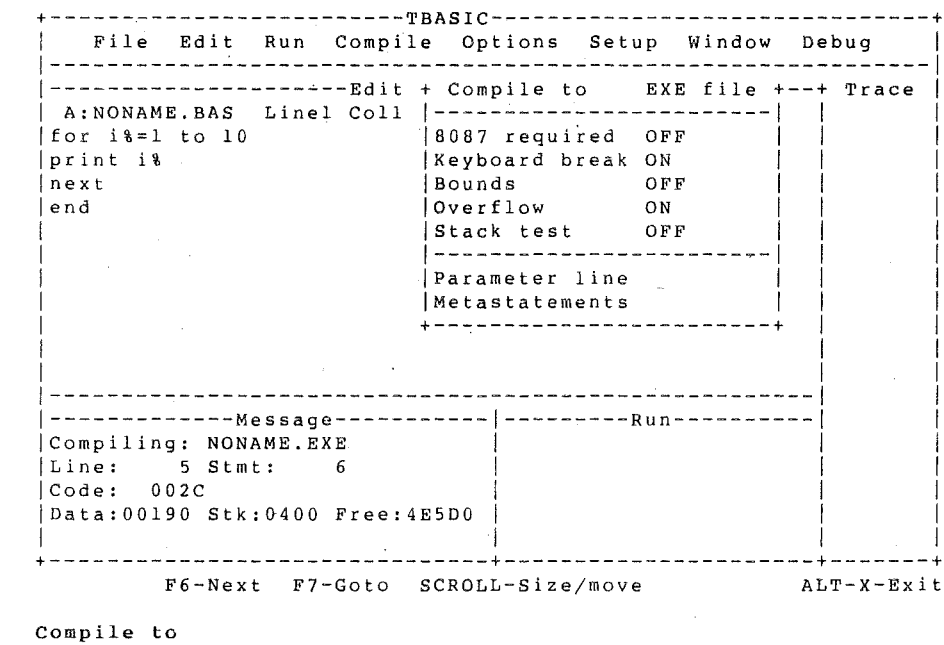
Der resultierende Objektcode wird abhaengig vom Stand des Schalters **Compile to** im Menue **Options** entweder im Speicher des Computers (Standardvorgabe) abgelegt oder auf der Diskette als Datei mit dem Dateityp .EXE bzw. .TBC geschrieben.

Waehrend des Compilierens wird die Anzahl der uebersetzten Zeilen im Fenster **Message** ausgegeben, der Compiler zaehlt dabei in Schritten zu je zwanzig Zeilen. Ein Druck auf <CTRL-BREAK> bricht den Uebersetzungsvorgang ab.

Stellt der Compiler einen Fehler innerhalb des Quelltextes fest, aktiviert er den Editor und setzt den Kursor auf die entsprechende Stelle. Die Fehlermeldung erscheint in der Statuszeile des Editors. Sie wird nach dem ersten Tastendruck automatisch geloescht. Nach der Korrektur sollte der Editor mit <ESC> verlassen und der Compiler erneut aufgerufen werden, solange, bis der Quelltext frei von syntaktischen Fehlern ist.

2.3.5. Menue Options

Die Eingabe des Befehls **O** innerhalb des Hauptmenues fuehrt in das Menue **Options**, in dem eine Reihe von Schaltern fuer den Compiler gesetzt werden koennen. Saemtliche Schalter dieses Menues sind "global" - sie gelten fuer die gesamte Uebersetzung eines Programms, solange sie nicht durch Compiler-Befehle im Quelltext ausser Kraft gesetzt werden.



Compile to
Ueber diese Auswahl wird ein weiteres Menue erreicht, in dem das Ziel des Compilierens bestimmt werden kann. Es bietet drei Moeglichkeiten, zwischen denen auf dieselbe Art gewaehlt werden kann wie bei den anderen Menues (Eingabe des hervorgehoben dargestellten Buchstabens oder Bewegen des Kursors und <ENTER>):

1. **Memory** (die Standardvoreinstellung von TBASIC): Der Objektcode wird im Hauptspeicher des Computers abgelegt. Diese Einstellung eignet sich am besten fuer das Austesten von Programmen und zur Fehlerkorrektur, weil sie die schnellstmoegliche Uebersetzung gewaehrleistet.

2. **Executable File:** Der Objektcode wird in Form einer direkt ausfuehrbaren Datei (.EXE) auf die Diskette geschrieben. In den meisten Faellen wird diese Einstellung erst dann benutzt, wenn ein Programm komplett ausgetestet ist. Einige Moeglichkeiten wie z.B. das Verbinden mehrerer Programmteile mit CHAIN lassen sich allerdings nur mit .EXE-Dateien ausprobieren.

3. **Chain File:** Der Objektcode wird in Form einer Datei mit dem Dateityp .TBC auf die Diskette geschrieben. Diese Dateien enthalten nicht die Laufzeitbibliothek von TBASIC und belegen deshalb wesentlich weniger Platz auf der Diskette. Gestartet und ausgefuehrt werden koennen sie nur mit den Kommandos RUN oder CHAIN von einem TBASIC-Programm, das als .EXE-Datei compiliert worden ist.

Die restlichen Punkte des Menues **Options** enthalten verschiedene Schalter, mit denen die Codeerzeugung des Compilers beeinflusst werden kann. Die Standardvorgabe fuer jeden dieser Schalter ist **OFF** - auf diese Weise wird der schnellste und kompakteste Code erzeugt.

Alle in den folgenden Abschnitten beschriebenen Schalter sind **Umschalter** - sie wechseln jedesmal ihren Zustand (von OFF auf ON bzw. von ON auf OFF), wenn sie angesprochen werden.
Die Auswahl geschieht auch hier wieder ueber die Eingabe des hervorgehobenen dargestellten Zeichens oder ueber die Kursorsteuertasten und <ENTER>.

8087 required
Wird dieser Schalter (durch die Eingabe von 8) auf ON gesetzt, erzeugt der Compiler direkte Anweisungen fuer den mathematischen Coprozessor, anstatt entsprechende Unterprogramme der Laufzeitbibliothek aufzurufen. Die Ausfuehrung rechenintensiver Programme wird dadurch erheblich beschleunigt. Damit erzeugte Programme laufen jedoch nur auf einem Computer, bei dem dieser Coprozessor installiert ist.

Ist der Schalter auf OFF gesetzt, nimmt ein uebersetztes Programm bei mathematischen Operationen eine Pruefung vor. Falls ein Coprozessor vorhanden ist, wird er benutzt. Ansonsten werden diese Operationen ueber Unterprogramme ausgefuehrt.

Keyboard break
Ist dieser Schalter auf ON gesetzt, kann die Ausfuehrung eines Programms durch <CTRL-BREAK> abgebrochen werden (CTRL-C hat keine Wirkung).
<CTRL-BREAK> zeigt erst dann eine Wirkung, wenn das Programm eine Ausgabe auf dem Bildschirm oder Drucker (aber nicht Grafik) vornimmt oder eine Eingabe ueber die Tastatur erwartet. Befehlsfolgen, die ohne Ein- und Ausgabe ablaufen, koennen grundsaetzlich nicht unterbrochen werden. (Abbruch ist nur durch einen Neustart des Computers moeglich).

Bounds
Ist dieser Schalter auf ON gesetzt, dann erzeugt der Compiler einen Code fuer die Pruefung von Feldgrenzen. Dadurch wird sichergestellt, dass Indizierungen beliebiger Art die Groesse der dazugehoerigen Variablen nicht ueberschreiten koennen.

Beispiel:

```
DIM FELD(50)
X = 65
Y = FELD(X)
```

Index-Ueberschreitungen durch Konstanten werden bereits vom Compiler abgefangen. Der folgende Quelltext erzeugt bei der Uebersetzung eine Fehlermeldung (unabhaengig davon, wie **Bounds** gesetzt ist):

```
DIM FELD(100)
Y = FELD(114)
```

Overflow
Wird der zulaessige Bereich fuer Integer-Variablen (-32768 bis +32767) durch eine Rechenoperation ueberschritten, findet ein Ueberlauf statt. Der Prozessor bzw. das Programm merken normalerweise nichts davon und arbeiten einfach mit einem Restwort im Integer-Bereich weiter.

Beispiel:

```
A% = 20000
B% = A% + A%
```

Das Ergebnis ueberschreitet die groesste zulaessige Integerzahl.

Wird diese Programm uebersetzt und **Overflow** ist auf ON gesetzt, wird bei der Ausfuehrung eine entsprechende Fehlermeldung ausgegeben.
Ist beim Uebersetzen **Overflow** auf OFF gesetzt, so wird der Variablen B% ein falscher Wert zugewiesen.

Das Pruefen auf Ueberlauf erfasst Operationen mit Integern und Langintegern, bei Realzahlen findet unabhaengig von Overflow immer eine Pruefung statt. Operationen mit Registern des Prozessors (REG-Befehl) sind etwas kritisch. Hier kann es sein, dass ein Ueberlauf trotz gesetzter Pruefung nicht entdeckt wird.

Stack test
Ist dieser Schalter auf ON gesetzt, erzeugt der Compiler Pruefroutinen fuer eventuelle Zusammenstoesse des Stacks mit dem Datenbereich, die bei jedem Aufruf einer Prozedur oder Funktion benutzt werden.
Um mehr Platz fuer den Stack freizuhalten, sollte der Compiler-Befehl **$STACK** verwendet werden (siehe Kapitel 4).

Die letzten beiden Wahlmoeglichkeiten des Menues **Options** sind keine Umschalter.

Parameter line
Ueber diese Wahlmoeglichkeit kann ein Aufruf des Programms von der DCP-Kommando-Ebene mit zusaetzlichen Parametern simuliert werden. TBASIC erwartet die Eingabe von bis zu 80 Zeichen, die das Programm beim naechsten Start als COMMAND$ lesen kann. Die Eingabe ist mit <ENTER> abzuschliessen. Mit <ESC> erfolgt die Rueckkehr ins Hauptmenue.

Metastatements
Die letzte Wahlmoeglichkeit von **Options** fuehrt zu einem eigenen Menue, dessen einzelne Punkte wiederum durch die Eingabe des hervorgehoben dargestellten Zeichens oder durch die Kursorsteuertasten zusammen mit <ENTER> gewaehlt werden koennen.

Die einzelnen Wahlmoeglichkeiten haben folgende Bedeutungen:

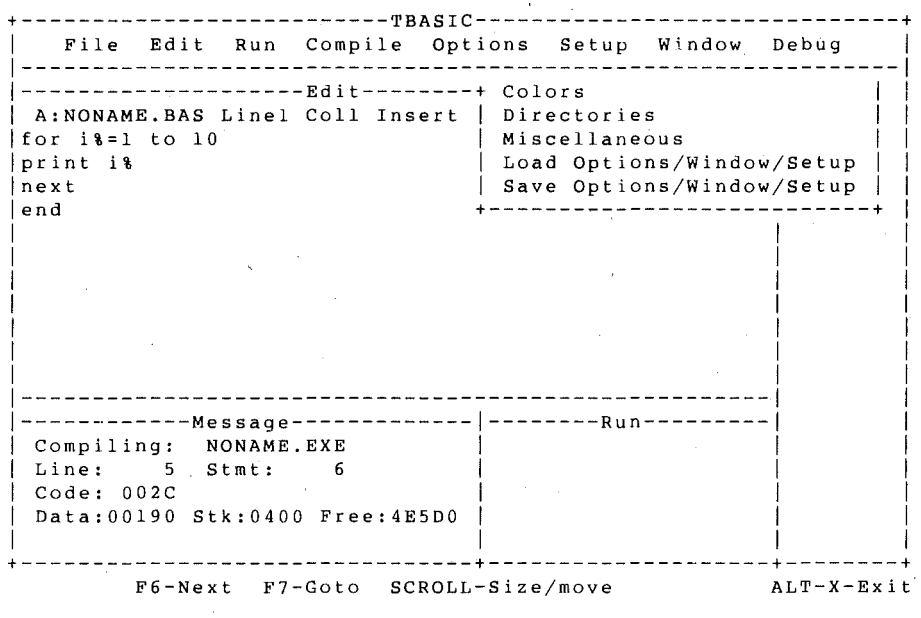
- **Stack size:** Setzt die Groesse des fuer den Stack verfuegbaren Bereichs. Die Angabe der momentan gesetzten Groesse erfolgt in Bytes, die Standardvorgabe ist 50300 Byte. Nach Betaetigen von **S** kann ein Neufestlegen dieser Groesse erfolgen, zulaessige Werte liegen im Bereich von 50300 bis 57FFF Byte (vgl. Compiler Befehl $STACK in Kapitel 4).

- **Music buffer:** Setzt die Groesse des Puffers fuer das (als Hintergrundprogramm laufende) Erzeugen von Toenen und Geraeuschen. Die Standardvorgabe ist 32 Noten (entspricht 256 Byte). Werden im Programm die Befehle SOUND und PLAY nicht benutzt, koennen diese Werte auf Null zurueckgesetzt und so die Laenge des Programms entsprechend verkuerzt werden. Die maximale Groesse dieses Puffers betraegt 4096 Noten (= 32768 Byte). Die Beschreibung des Compiler-Befehls $SOUND in Kapitel 4 enthaelt weitere Einzelheiten.

- **Communication:** Zeigt ein weiteres Untermenue an, in dem die Puffergroesse fuer die DCP-Geraeteeinheiten COM1: und COM2: gesetzt werden koennen. Die Standardvorgabe ist 256 Byte fuer jeden der beiden Puffer, zulaessige Werte fuer eine Eingabe gehen von 0 bis 32 KByte (vgl. Compiler-Befehl $COMn in Kapitel 4).

2.3.6. Menue Setup

Der Befehl **S** bzw. Bewegen des hervorgehoben dargestellten Teils des Balkens auf **Setup** und <ENTER> aktiviert das Menue **Setup**. Es enthaelt verschiedene Moeglichkeiten zum Veraendern der Standardvorgaben von TBASIC. Einmal festgelegte Voreinstellungen koennen permanent in einer Datei mit dem Dateityp .TB oder TBASIC.EXE selbst gespeichert werden. Optionen des Compilers werden ebenfalls erfasst.



Colors
Die Auswahl von **Colors** fuehrt in ein ganzes System weiterer Menues und Untermenues (vier Ebenen sind es insgesamt), ueber die das Erscheinungsbild des Menues, der Fenster fuer Meldungen des Systems und der vier Fenster von TBASIC veraendert werden koennen.

Beispiel: Die Auswahl von **Menue** fuehrt zu der Anfrage, ob die Menues selbst (**Main pull-down**), das Untermenue der ersten Ebene (**First pop-up**), ein eventuelles Untermenue der zweiten Ebene (**Second pop-up**) oder der dritten Ebene (**Third pop-up**) in seiner Farbe veraendert werden soll. Nachdem eine Auswahl getroffen wurde, muss angegeben werden, welcher Teil des entsprechenden Menues veraendert werden soll: den normalerweise Normal oder Invers dargestellten Text, Grossbuchstaben (**Capital letter**), hervorgehoben dargestellte Grossbuchstaben (**Highlighted caps**), den Rand (**Border**), den Titel (**Title**) - oder ob alle Aenderungen wieder rueckgaengig gemacht werden sollen.

Wurde eine Moeglichkeit ausgewaehlt, z.B. **Main pull-down Title**, dann erscheint auf dem Bildschirm eine Farbtabelle, aus der mit Hilfe der Kursorsteuertasten eine Kombination von Vorder- und Hintergrundfarben ausgewaehlt werden kann. Das momentan aktive Fenster zeigt jeweils, wie das Ergebnis auf dem Bildschirm aussieht.

Directories
Auch hier erscheint ein weiteres Menue auf dem Bildschirm - mit ihm koennen die Suchwege fuer die (maximal) drei von TBASIC verwendeten Directories bestimmt werden.

Die Wahlpunkte haben folgende Bedeutungen:

- **Executable:** In das hier angegebene Directory schreibt der Compiler Dateien vom Typ .EXE und .TBC ein, die als Ergebnis von Uebersetzungen erzeugt werden.

- **Include:** In dem hier angegebenen Directory sucht der Compiler Quelltexte , deren Namen innerhalb des "Main File" mit $INCLUDE angegeben werden.

- **TURBO:** Setzt den Suchweg fuer die von TBASIC selbst verwendeten Dateien, also fuer TBASIC.EXE und gespeicherte Konfigurationen (.TB).

Wird einer der drei Punkte gewaehlt, muss ein Suchweg angegeben werden, der den Regeln von DCP entspricht bzw. die Taste <ENTER> gedrueckt werden. In diesem Fall erwartet TBASIC die entsprechenden Dateien im Standard-Directory. Durch einen Druck auf <ESC> anstelle einer Eingabe erfolgt die Rueckkehr in das Menue **Directories**, ohne dass der entsprechende Suchweg veraendert wird.

Miscellaneous
Dieses durch den Befehl **M** aufgerufene Untermenue ermoeglicht das Setzen von zwei speziellen Parametern:

- **Auto save edit:** Ist dieser Umschalter auf ON gesetzt, speichert TBASIC den Quelltext vor jedem Start eines Programms, falls er seit dem letzten Speichern veraendert wurde. Die Standardvorgabe ist OFF.

- **Backup source:** Normalerweise erhaelt TBASIC die jeweils vorletzte Version eines Quelltextes - die entsprechende Datei bekommt den Dateityp .BAK angehaengt. Wird dieser Umschalter auf OFF gesetzt, dann wird die Arbeitsdatei beim Speichern einfach mit den neuen Daten ueberschrieben. Die Standardvorgabe fuer **Backup source** ist ON.

Saemtliche Operationen des Compilers koennen zusammen mit den von **Setup** festgelegten Farben, Suchwegen und restlichen Einstellungen entweder in einer eigenen Konfigurationsdatei oder innerhalb von TBASIC.EXE selbst gespeichert werden. Es kann beispielsweise eine Datei mit einem vergroesserten Trace-Fenster erstellt werden, in der saemtliche Compiler-Pruefungen auf ON gesetzt sind und sie zur Entwicklung von Programmen benutzen. Eine andere Datei enthaelt die Konfiguration zum Uebersetzen von fertig ausgetesteten Programmen (Pruefen auf OFF, Compile to Executable File usw.)

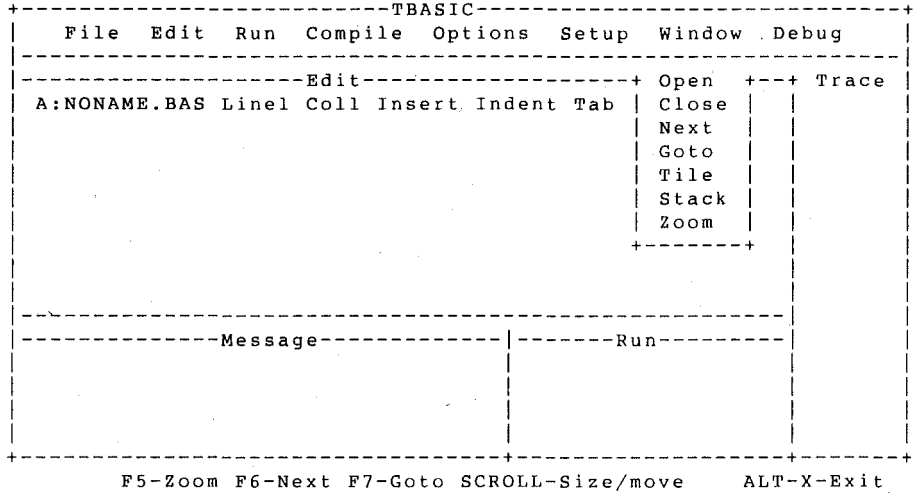
Mit den beiden letzten Wahlmoeglichkeiten des Menues **Setup** koennen Konfigurationsdateien geladen bzw. gespeichert werden.

Load Options/Setup
Laedt eine Konfigurationsdatei (.TB) mit dem angegebenen Namen und setzt saemtliche Optionen und Parameter von TBASIC entsprechend.

Save Options/Setup
Speichert die momentan gesetzten Optionen und Parameter in einer Konfigurationsdatei mit dem angegebenen Namen. Es kann ein beliebiger Name benutzt werden, solange er den Regeln von DCP entspricht. Der Dateityp .TB wird automatisch angehaengt. Wird kein Name angegeben, verwendet TBASIC die Vorgabe TBCONFIG.TB. Wird der Name TBASIC.EXE angegeben, erscheint die Anfrage, ob TBASIC permanent veraendert werden soll. Wird diese Anfrage bestaetigt, wird eine neue Version von TBASIC.EXE auf die Diskette geschrieben.

2.3.7. Menue Windows

Durch den Befehl **W** bzw. Bewegen des hevorgehoben dargestellten Balkens mit den Kursortasten und <ENTER> wird das Menue **Windows** aufgerufen, dessen einzelne Wahlpunkte in den folgenden Abschnitten beschrieben sind. Alle Kommandos, ausser **Open**, beziehen sich auf das momentan aktive Fenster.



Open
Mit diesem Kommando wird ein Fenster eroeffnet und gleichzeitig zum aktiven Fenster gemacht. Auf dem Bildschirm erscheint ein Untermenue, in dem zwischen **Edit, Run, Message** und **Trace** gewaehlt werden kann. Wurde das entsprechende Fenster zuvor mit **Close** vom Bildschirm entfernt, wird es wieder sichtbar. Ansonsten aktiviert **Open** nur das gewaehlte Fenster. Es bekommt eine doppelt ausgezogene Linie und erscheint als vorderstes auf dem Bildschirm.

Close
Entfernt das momentan aktive Fenster vom Bildschirm und "versteckt" es. Schreiben in dieses Fenster ist allerdings nach wie vor moeglich.

Next
Macht das momentan aktive Fenster inaktiv und setzt das naechste Fenster aktiv. Dies entspricht also einem **Close** und einem **Open** fuer das naechste Fenster. Ist das momentan aktive Fenster das letzte der Liste, wird als naechstes das erste Fenster der Liste wieder aktiv gesetzt. Ein Druck auf die Funktionstaste F6 innerhalb eines Fensters oder auf der Ebene des Hauptmenues hat die gleiche Wirkung.

Goto
Mit diesem Kommando erfolgt direkt der Uebergang in das (evt. vorher durch **Open** gesetzte) aktive Fenster.

Tile
Setzt alle aktiven Fenster auf die gleiche Groesse und verteilt sie gleichmaessig nebeneinander auf dem Bildschirm. Ein Druck auf F6 aktiviert das naechste Fenster.

Stack
Erweitert alle aktiven Fenster auf die maximale Groesse und setzt sie hintereinander auf dem Bildschirm. Das momentan aktive Fenster kommt auf die vorderste Position und ist als einziges vollstaendig zu sehen. Ein Druck auf F6 bringt jeweils das hinterste Fenster nach vorn und aktiviert es.

Zoom
Die Auswahl dieses Kommandos setzt das momentan aktive Fenster auf die maximal moegliche Groesse und wirkt nur, wenn es sich bei diesem Fenster um **Edit** oder **Run** handelt. Die Funktionstaste F5 hat die gleiche Wirkung. Soll das Fenster **Run** umgeschaltet werden, waehrend ein Programm laeuft, muss <ALT-F5> benutzt werden.

2.3.8. Menue Debug

Die Eingabe des Befehls **D** bzw. Bewegen des hervorgehoben dargestellten Balkens mit den Kursorsteuertasten auf **Debug** und <ENTER> bringt ein Menue auf den Bildschirm, das die Wahlmoeglichkeiten **Trace** und **Run-time error** enthaelt.

Trace
Mit diesem Umschalter kann festgelegt werden, ob die Ausfuehrung des gesamten Programms von TBASIC verfolgt werden soll. Die Wirkung von **Trace** bezieht sich immer auf das gesamte Programm. Durch die Compiler-Befehle TRON und TROFF im Quelltext wird sie ausser Kraft gesetzt.

Run-time Error
Dieses Kommando ist hauptsaechlich fuer die Fehlersuche in Programmen vorgesehen, die in der Form von .EXE bzw. .TBS-Dateien gelaufen sind. In diesen Faellen wird bei einem Laufzeitfehler nur der Stand des Programmzaehlers ausgegeben. Solange ein Programm innerhalb der Programmierumgebung von TBASIC ausgetestet wird, laeuft die Zuordnung der entsprechenden Stelle im Quelltext bei einem Laufzeitfehler automatisch ab.

Nach der Auswahl von Run-time Error durch Eingabe eines **R** erwartet TBASIC die Eingabe des Programmzaehler-Standes, d.h. des als **pgm-ctr** ausgegebenen Wertes nach dem Abbruch des Programms. Danach wird die Fehlerstelle im Quelltext lokalisiert und der Editor aufgerufen. Der Kursor ist entsprechend gesetzt.

Die Zuordnung "Programmzaehler/Fehlerstelle" geschieht ueber ein erneutes Compilieren des Quelltextes und einen Vergleich der dabei erzeugten Adressen. Ein Beispiel dazu:

```
X = 256
PRINT CHR$(X)
```

Dieses Programm ist zwar syntaktisch korrekt und wird deshalb ohne Beanstandung vom Compiler uebersetzt. Der Versuch, ein Zeichen mit dem ASCII-Code 256 auszugeben, endet aber mit der Fehlermeldung:

```
Fehler 5 Illegaler Funktionsaufruf, pgm-ctr: 29
```

Die Ausgabe der Fehlermeldung im Klartext erfolgt aber nur innerhalb der Programmierumgebung von TBASIC. Wird dieses Programm als .EXE-Datei ausgefuehrt, wird lediglich eine Fehlernummer und der Stand des Programmzaehlers ausgegeben. Mit **Run-time error** kann die Stelle des Fehlers im Quelltext in beiden Faellen lokalisiert werden.

3. Allgemeines, Sprachelemente

3.1. Zeilenaufbau und Beschreibungsform der Sprache

3.1.1. Zeilenaufbau

Ein TBASIC-Programm besteht aus einer oder mehreren Zeilen Quelltext. Jede dieser Zeilen kann dabei eines der folgenden Formate haben:

```
[Zeilennummer:] Anweisung [:Anweisung]...['Kommentar]
Marke:
$Compiler-Befehl
```

Zeilennummer: ist eine Integerzahl im Bereich von 0 bis 65535 und hat dieselbe Funktion wie eine Marke.

Zeilennummern koennen frei mit Marken gemischt werden und duerfen in verschiedenen Programmteilen sogar ganz fehlen. Eine bestimmte numerische Reihenfolge muss ebenfalls nicht eingehalten werden. Es gibt lediglich drei Beschraenkungen:

1. Innerhalb eines Programms darf ein und dieselbe Zeilennummer nur einmal vorkommen; dasselbe gilt fuer Marken.

2. Eine Zeile darf entweder mit einer Marke oder mit einer Zeilennummer beginnen - beides zusammen ist nicht erlaubt.

3. Bei der Verfolgung des Programmablaufs mit **Trace** werden nur Zeilennummern angezeigt.

Anweisung: bezeichnet die einzelnen Kommandos und Anweisungen von TBASIC. Aus ihnen besteht das eigentliche Programmm (siehe Kapitel 4). Eine Zeile kann eine oder mehrere Anweisungen enthalten, die durch Doppelpunkt voneinander getrennt sind. Die maximale Anzahl der Zeichen darf 249 nicht ueberschreiten.

Kommentar: ist eine beliebige Folge von Zeichen und muss sich hinter eventuellen Anweisungen innerhalb der Zeile befinden. Durch ein Hochkomma (') wird er vom restlichen Text der Zeile abgetrennt. Kommentar kann auch mit einem REM eingeleitet werden. Dabei muss zwischen eventuellen Anweisungen und dem REM ein Doppelpunkt gesetzt werden.

Marke: gilt als Startpunkt fuer die Anweisungen der folgenden Zeile(n) und muss auf einer eigenen Zeile stehen.
Innerhalb derselben Zeile duerfen keine Anweisungen folgen, ein Kommentar ist allerdings erlaubt. Marken muessen grundsaetzlich mit einem Buchstaben beginnen, auf den eine beliebige Anzahl von Buchstaben und Ziffern folgen kann (entsprechend des zugelassenen Zeichensatzes). Zwischen Gross- und Kleinbuchstaben unterscheidet der Compiler nicht.

Eine Marke muss grundsaetzlich mit einem Doppelpunkt abgeschlossen werden.

Compiler-Befehle: haben mit dem eigentlichen BASIC-Programm nichts zu tun. Sie beginnen grundsaetzlich mit dem Waehrungszeichen/Dollarzeichen und geben dem Compiler waehrend der Uebersetzungszeit entsprechende Anweisungen.

Ein Beispiel fuer einen Compiler-Befehl, der sich nicht ueber das Menue **Options** setzen laesst, ist $INCLUDE. Der Compiler fuegt waehrend der Uebersetzung am Punkt dieses Befehls den Inhalt einer weiteren Datei in den Quelltext ein. Im Gegensatz zu anderen BASIC-Compilern erwartet TBASIC Compiler-Befehle frei im Quelltext, sie muessen allerdings auf einer eigenen Zeile stehen (Kommentare innerhalb dieser Zeile sind erlaubt). Eine Zeile darf jeweils nur einen Compiler-Befehl enthalten.

Im Gegensatz zum BASIC-Interpreter ignoriert der Compiler alle Leerzeichen und -zeilen sowie alle Kommentare.

Somit wirkt sich eine gute Formatierung und Kommentierung des Quelltextes nicht auf die Laufzeit des Programms aus.
Soll eine Anweisung auf der naechsten Zeile fortgesetzt werden, so sollte als letztes Zeichen der Zeile ein Unterstrich (_) eingegeben werden.
Der Compiler betrachtet dann die naechste Zeile als luckenlose Fortsetzung.

Beispiele fuer gueltige Programmzeichen:

```
SUM:                                'nur eine Marke

x=x+1                               'Anweisung

30 y=y+10                           'Zeilennummer und Anweisung

$INCLUDE "DATEN.ASC"                'Compilerbefehl

FIELD #1, 30 AS NAM$,_              'Fortsetzen der Zeile mit
          30 AS ADR$,_               Unterstrich
          30 AS BER$,_
          30 AS BETR$
```

3.1.2. Beschreibungsform der Sprache

Die Syntax der im Kapitel 4 beschriebenen Kommandos, Anweisungen und Funktionen ist nach folgenden Konventionen zu nutzen:

- Woerter in Grossbuchstaben sind Schluessselwoerter und muessen komplett eingegeben werden (Klein- und Grossbuchstaben sind moeglich).

- Angaben in Kleinbuchstaben sind Parameter und Argumente, die vom Benutzer definiert und eingegeben werden muessen.

- Angaben in eckigen Klammern ([]) sind wahlfrei.

- Drei Punkte hintereinander bedeutet, dass eine Angabe beliebig oft wiederholt werden kann.

- Jede Interpunktion wie z.B. Klammern, Semikolon, Komma, Hochkomma oder Gleichheitszeichen muss an der angezeigten Stelle eingefuegt werden.
  Ausnahme: eckige Klammern.

3.2. Zeichensatz

Wie jede Hochsprache stellt TBASIC eine Reihe von einzelnen Bausteinen fuer das Erstellen von Programmen zur Verfuegung. Damit der Compiler zwischen Variablennamen und Programmierbausteinen unterscheiden kann, werden verschiedene Zeichen und Zeichenfolgen als "reserviert" betrachtet - sie duerfen nicht (oder nur in bestimmtem Kontext) verwendet werden.

Die Grossbuchstaben **A...Z** sowie die Kleinbuchstaben **a..z** und die Ziffern **0...9** koennen als Bezeichner fuer Variablen, Konstanten, Prozeduren und Funktionen verwendet werden. Die Umlaute und das "ss" sind nicht verwendbar.

Die Ziffern **0...9**, die Symbole **Punkt(.)**, **Plus(+)**, und **Minus(-)** sowie die Buchstaben **E**, **e**, **D** und d sind gueltige Zeichen innerhalb von numerischen Konstanten.

Die folgenden Zeichen werden von TBASIC in spezieller Weise verstanden, d.h. sind "reserviert":

Zeichen	Beschreibung / Funktion
=	Gleichheitszeichen (Zuordnungen, Vergleichsoperator)
+	Pluszeichen (Addition, Verbindung von Teilzeichenketten)
-	Minuszeichen (Subtraktion und Negation)
*	Stern (Multiplikation)
/	Schraegstrich (Division)
\	umgekehrter Schraegstrich (Integer-Division)
^	Caret (Potenzierung)
%	Prozentzeichen (Festlegung auf den Typ Integer und Einleitung fuer benannte Integer-Konstanten)
&	Ampersand (Festlegung auf den Typ Langinteger und Einleitung fuer nichtdezimale Konstanten)
!	Ausrufezeichen (Festlegung auf den Typ Real, einfache Genauigkeit)
#	Nummernzeichen (Festlegung auf den Typ Real, doppelte Genauigkeit)
$	Dollarzeichen/Waehrungszeichen (Festlegung auf den Typ Zeichenkette, Einleitung fuer Compiler-Befehle)
()	Klammern (Argumente fuer Prozeduren und Funktionen, Vorrang bei arithmetischen Ausdruecken)
	Leerzeichen (Trennung)
,	Komma
'	Hochkomma (Einleitung fuer Kommentare)
;	Semikolon
:	Doppelpunkt (Trennung von Anweisungen)
?	Fragezeichen (Abkuerzung fuer PRINT)
<>	spitze Klammern (Vergleichs-Operatoren)
"	Anfuehrungszeichen (Trennung fuer Zeichenketten)
_	Unterstrich (Zeilen-Verlaengerung)

3.3. Numerische Datentypen

TBASIC unterstuetzt bei der Verarbeitung von numerischen Werten vier unterschiedliche Variablentypen:

Integer, Langinteger, einfachgenaue und doppeltgenaue Realzahlen.

- Integer
  Einfachster Datentyp.
  Operationen mit ihm laufen mit der groessten Geschwindigkeit ab.
  Ganze Zahl (ohne Dezimalpunkt und ohne Nachkommastellen) im Bereich von -32768 bis +32767
  Speicherung in 2 Byte.
  Ist der Schalter **Overflow** im Menue **Options** auf ON gesetzt, erzeugt der Compiler zusaetzlichen Code, um eventuelle Ueberlaeufe bei Operationen mit Integer-Werten abzufangen.

- **Langinteger**
  Ganze Zahl im Bereich von -2 hoch 33 bis +2 hoch 33 (-2147483648... +2147483647).
  Speicherung in 4 Byte.
  Verarbeitungsgeschwindigkeit gegenueber Daten vom Typ Integer ist geringer.

- **Realzahlen einfacher Genauigkeit**
  Positive oder negative reelle Zahlen (d.h., sie koennen einen Dezimalpunkt enthalten) im Bereich von 10-38 bis 10+38.
  Speicherung in 4 Byte.
  Die Speicherung von einfachgenauen Realzahlen erfolgt mit einer Genauigkeit von 6 Dezimalstellen.

- **Realzahlen doppelter Genauigkeit**
  Positive oder negative reelle Zahlen im Bereich von 10-308 bis 10+308.
  Speicherung in 8 Byte.
  Die Speicherung erfolgt mit einer Genauigkeit von 16 Dezimalstellen.

Hinweis:

TBASIC benutzt zur Speicherung von Realzahlen das IEEE-Format. Dieses Format unterscheidet sich vom Format der Speicherung von Realzahlen im BASIC-Interpreter BASI. Damit Werte von Dateien gelesen werden koennen, die mit BASI erstellt wurden, bzw. BASI-kompatible Dateien geschrieben werden koennen, stehen spezielle Konvertierungsfunktionen (CVMS, CVMD, MKMS$, MKMD$) zur Verfuegung (siehe Kapitel 4).

3.4. Konstanten, Variablen, Felder

Ein in TBASIC geschriebenes Programm arbeitet mit zwei verschiedenen Formen von Daten, naemlich **Konstantan** und **Variablen**. Der Wert einer Variablen kann sich waehrend der Laufzeit eines Programms aendern, der einer Konstanten dagegen nicht - er wird waehrend des Compilierens berechnet bzw. eingesetzt und bleibt waehrend der gesamten Laufzeit unveraendert.

3.4.1. Konstanten

TBASIC kennt drei verschiedene Arten von Konstanten:

Numerische Konstanten, Zeichenkettenkonstanten, Benannte Konstanten

Zeichenkettenkonstanten

Eine Zeichenkettenkonstante ist eine Folge von Zeichen, die durch Anfuehrungszeichen begrenzt ist, z.B.:

```
"Das ist eine Zeichenkette"
"012345"
```

Folgt innerhalb einer Zeile auf eine Zeichenketten-Konstante kein weiterer Text, dann kann das abschliessende Anfuehrungszeichen weggelassen werden.

Numerische Konstanten

Numerische Konstanten stellen Zahlenwerte dar und bestehen aus den Ziffern 0...9 und einem eventuellen Dezimalpunkt. Negative Werte werden durch ein vorangestelltes Minuszeichen (-) gekennzeichnet , ein Pluszeichen (+) bei positiven Werten ist zulaessig, aber optional. TBASIC bestimmt das Speicherformat aus der vorgegebenen Genauigkeit - von ihr haengt es ab, ob die Konstante bei einer Zuweisung als Integer, Langinteger, einfach oder doppeltgenauer Wert behandelt wird. Fuer diese automatische Definition gelten die folgenden Bedingungen :

- Enthaelt die Konstante keinen Dezimalpunkt und liegt ihr Wert im Bereich von -32768 bis +32767, wird der Typ **Integer** benutzt.

- Enthaelt die Konstante keinen Dezimalpunkt und liegt ihr Wert ausserhalb der Integergrenzen, aber innerhalb des Bereichs von -231 bis +231, wird der Variablen der Typ **Langinteger** zugeordnet.

- Enthaelt die Konstante einen Dezimalpunkt und maximal sechs Ziffern, wird eine **einfach genaue Realzahl** benutzt.

- Enthaelt die Konstante entweder einen Dezimalpunkt und mehr als sechs Ziffern oder einen Wert ausserhalb des Langinteger-Bereichs, nimmt TBASIC eine **doppelt genaue Realzahl** an.

Integer-Konstanten in nicht-dezimaler Notation

TBASIC ermoeglicht die direkte Arbeit mit Hexadezimal- (Basis 16), Oktal- (Basis 8) und Binaerzahlen (Basis 2). Diese alternativen Basen sind nur fuer einfache Integerwerte definiert. Die zulaessigen Werte liegen also im Bereich von 0 bis 65535.

- **Hexadezimale Konstanten** bestehen aus bis zu vier aufeinanderfolgenden Ziffern (0...9) und Buchstaben (A...F). Sie werden durch Voranstellen von &H gekennzeichnet.Der zulaessige Wertebereich geht von &H0000 bis &HFFFF.

- **Oktale Konstanten** bestehen aus bis zu sechs aufeinanderfolgenden Ziffern (0...7) und werden durch Voranstellen eines &O (alternativ: &O oder einfach &) gekennzeichnet. Der zulaessige Wertebereich geht von &O000000 bis &O777777.

- **Binaere Konstanten** bestehen aus bis zu sechzehn aufeinanderfolgenden Ziffern (0...1) und werden durch Voranstellen eines &B gekennzeichnet. Der zulaessige Wertebereich geht von &B0000 0000 0000 0000 bis &B1111 1111 1111 1111.

Benannte Konstanten

Mit TBASIC ist es moeglich, anstelle eines konstanten Wertes einen Namen zu verwenden- etwa in derselben Form wie durch die CONST-Deklaration von Pascal. Diese Moeglichkeit stellt eine Erweiterung der Sprachdefinition von BASIC dar und ist nur fuer Integer-Konstanten zulaessig.

Der Name einer Konstanten beginnt mit einem Prozentzeichen (%). Im Gegensatz zu einer Variablen darf einer Integer-Konstanten nur ein einziges Mal in einem Programm ein Wert zugewiesen werden. Der Wert selbst muss eine numerische Konstante sein - Variablenzuweisungen oder arithmetische Ausdruecke sind nicht erlaubt.

Beispiel:

```
%MAX = 1000                    definiert eine Konstante
IF x% > %MAX THEN...
PRINT %MAX, MAX%
```

Der PRINT-Befehl in der dritten Zeile gibt zuerst den Wert der benannten Konstanten %MAX und danach den Wert der Variablen MAX% aus.

3.4.2. Variablen

Unter diesem Begriff wird ein Name fuer einen numerischen Wert oder eine Zeichenkette verstanden. Im Gegensatz zu einer Konstanten kann ihr Wert innerhalb eines Programms beliebig oft wechseln. Variablennamen muessen (genau wie Marken) mit einem Buchstaben beginnen, auf den eine beliebige Anzahl von Buchstaben und Ziffern folgen darf. Ein Variablenname kann theoretisch bis zu 255 Zeichen lang sein (es sind alle Zeichen signifikant). Zwischen Gross- und Kleinbuchstaben wird nicht unterschieden. Die Umlaute und das "ss" sind innerhalb von Variablen nicht erlaubt.

Die Laenge eines Namens bei TBASIC hat nicht den geringsten Einfluss auf die Ausfuehrungsgeschwindigkeit oder den Speicherplatzbedarf des Programmcodes.

Es sind fuenf verschiedene Arten von Variablen moeglich:

Zeichenkette, Integer, Langinteger, einfach- und doppeltgenaue Realzahlen.

Der Typ einer Variablen wird entweder ueber eine Konstanten-Zuweisung oder ueber ein spezielles Zeichen im Namen bestimmt. Diese speziellen Typendefinitionszeichen werden an das Ende des Variablennamens angefuegt und sind folgende:

$ Zeichenkettenvariable
% Integer-Variable
& Langinteger-Variable
! Real-Variable einfacher Genauigkeit
Real-Variable doppelter Genauigkeit

Enthaelt der Variablenname keines dieser Typdefinitionszeichen, so wird standardmaessig eine Real-Variable einfacher Genauigkeit angenommen.

3.4.3. Felder

Ein Feld ist eine Gruppe von gleichartigen Variablen, die unter einem gemeinsamen Namen zusammengefasst sind. Jeder Einzelwert eines Feldes wird als **Element** bezeichnet. Variablen-Felder von TBASIC koennen entweder numerische Werte oder Zeichenketten enthalten und in jedem Ausdruck eingesetzt werden, in dem auch eine einfache Variable gleichen Typs zulaessig ist.

Beim Start eines Programms werden saemtliche numerischen Feld-Elemente auf den Wert 0 gesetzt, die Elemente von Zeichenkettenfeldern bekommen die Nullzeichenkette ("") zugewiesen. Die Erzeugung eines dynamischen Feldes mit der Anweisung DIM loescht ebenfalls saemtliche seiner Elemente.

Wenn sich das Programm selbst mit RUN neu startet, laeuft dieser Initialisierungsprozess erneut ab.

Mit der Anweisung DIM werden sowohl der Name und Typ eines Feldes als auch die Anzahl seiner Elemente festgelegt.
Die Anweisung DIM TABELLE(50) erzeugt eine Feld-Variable mit dem Namen **Tabelle**, die aus 51 Elementen vom Typ einfachgenauer Realzahlen besteht. Das erste Element hat die Nummer 0, das letzte die Nummer 50.

Maximal sind in einem (eindimensionalen) Feld 32768 Elemente moeglich. Diese Grenze kann aber nur mit Integern ausgenutzt werden - der maximale Speicherplatz, der einem Feld zugeordnet werden kann, betraegt 64 KByte.

Zugriff auf Feld-Elemente

Die einzelnen Elemente eines Feldes werden ueber ihre Nummer **indiziert**: Hinter dem Feldnamen folgt ein ganzzahliger Wert in Klammern, der die Nummer des entsprechenden Elementes bezeichnet. Mit Tabelle(4) wird z.B. das fuenfte Element, mit Tabelle(50) das letzte Element des Feldes **Tabelle** bezeichnet.

Es ist theoretisch moeglich, ein Feld ohne vorherige DIMensionierung zu verwenden. TBASIC erstellt beim ersten Zugriff auf ein nicht definiertes Feld automatisch ein Feld entsprechenden Typs mit 11 Elementen (Indexnummern von 0 bis 10). Diese Art ist jedoch nicht empfehlenswert. Es sollten alle Felder eines Programms explizit dimensioniert werden.
Standardmaessig hat das erste Element eines Feldes die Nummer 0, dies kann aber mit der Anweisung OPTION BASE oder einem zusaetzlichen Parameter fuer DIM geaendert werden.
Anstelle einer einfachen Obergrenze kann ein **Bereich** fuer die Indizierung definiert werden. Die Anweisung

```
DIM Jahr(1901:2000)
```

erzeugt ein Feld **Jahr**, das 100 Elemente im Format einfachgenauer Realzahlen enthaelt. Die Anweisung

```
DIM M(1:12,1:31)
```

erzeugt ein zweidimensionales Feld.

Durch diese Erweiterung der Anweisung DIM koennen Datenstrukturen wesentlich besser an die tatsaechliche Aufgabenstellung angepasst werden.

Zeichenkettenfelder

Die Elemente eines Zeichenkettenfeldes enthalten keine numerischen Werte, sondern Zeichenketten. In derselben Weise wie bei numerischen Feldern kann jedes einzelne Element als Variable (hier vom Typ Zeichenkette) betrachtet werden. Aus diesem Grund koennen sie eine unterschiedliche Laenge haben: von 0 bis 32767 Zeichen. Insgesamt stehen fuer Zeichenketten und Zeichenkettenfelder 64 KByte Speicherplatz zur Verfuegung.

Beispiel:

```
DIM  MON$(12)
MON$(1)="Januar"
MON$(2)="Februar"
MON$(3)="Maerz"
```

Mehrdimensionale Felder

Felder koennen eine oder mehrere **Dimensionen** haben. Das Maximum betraegt 8. Ein eindimensionales Feld wie Tabelle kann man als eine Liste einzelner Variablen betrachten, ein zwei-dimensionales Feld dagegen als Feld mit Spalten und Zeilen. Multidimensionale Felder sind auf diese Weise ebenfalls moeglich.

```
DIM a(10)              'ein eindimensionales Feld
DIM b(10,20)           'ein zweidimensionales Feld
DIM c(10,15,5)         'ein dreidimensionales Feld
DIM d(20,10,5,10)      'ein vierdimensionales Feld
```

Die maximale Anzahl von Elementen pro Dimension betraegt auch bei mehrdimensionalen Feldern 32768.

Ueberpruefung von Feldgrenzen

Ueberschreitungen der Feldgrenzen lassen sich unterteilen in statische Ueberschreitungen wie

```
DIM MON(12)
MON(13)=30
```

und Ueberschreitungen waehrend der Laufzeit eines Programms:

```
DIM MON(12)
a = 13
MON(a) = 30.
```

Die erste Art von Fehlern wird vom Compiler bereits waehrend des Uebersetzens durch eine entsprechende Fehlermeldung abgefangen. Da die im zweiten Beispiel gezeigte Befehlsfolge syntaktisch korrekt und der Wert der Variablen a waehrend der Uebersetzung nicht bekannt ist, wird dieses Programm ohne Beanstandung uebersetzt. Um diese Art von Fehlern abzufangen, muss der Umschalter **Bounds** im Menue **Options** auf ON gesetzt sein (siehe Kapitel 2).

Speichergrenzen fuer Felder

Aus Gruenden der Ausfuehrungsgeschwindigkeit und des Platzbedarfs fuer den Programmcode begrenzt TBASIC die Groesse eines einzelnen Feldes auf 64 KByte - allerdings kann ein Programm so viele Felder enthalten, wie tatsaechlich an Speicher verfuegbar ist. Die maximale Anzahl von Elementen innerhalb eines Feldes wird durch diese Begrenzung eine Funktion des Feldtypes:

Typ	Speicherplatzverbrauch
Integer	2 Byte pro Element -> max. 32768 Elemente
Laninteger	4 Byte pro Element -> max. 16384 Elemente
einfachgenau Real	4 Byte pro Element -> max. 16384 Elemente
doppeltgenau Real	8 Byte pro Element -> max. 8192 Elemente
Zeichenkette	4 Byte pro Element -> max. 16384 Elemente

Hinweis:

In einem Zeichenkettenfeld werden fuer jedes Element lediglich die Laenge und ein Zeiger auf den Zeichenketteninhalt gespeichert. Die Zeichenketteninhalte selbst sind in einem anderen Bereich (dem Zeichenkettenbereich) gespeichert, der eine Groesse von maximal 64 KByte hat.

Dynamische Belegung des Speichers

In allen vorhergehenden Beispielen wurde die Anweisung DIM grundsaetzlich zusammen mit einer Konstanten benutzt. Der Compiler bestimmt waehrend des Uebersetzens den Wert der Konstanten und reserviert eine entsprechende Menge an Speicherplatz. Diese Art der Feld-Erzeugung ist **statisch:** die Groesse des Feldes liegt bereits vor dem spaeteren Start des Programms fest. Zusaetzlich erlaubt TBASIC die Verwendung **dynamischer** Felder, also Felder, deren Groesse erst waehrend der Laufzeit des Programms festgelegt wird. Der durch ein dynamisch erzeugtes Feld belegte Speicherplatz kann zu einem spaeteren Zeitpunkt wieder freigegeben und fuer andere Zwecke verwendet werden.

Um ein Feld dynamisch zu erzeugen, muss eine Variable anstelle einer Konstanten zusammen mit der Anweisung DIM verwendet werden. Das entsprechende Feld wird dann erst zur Laufzeit des Programms erzeugt (Voraussetzung: es muss genuegend Speicherplatz vorhanden sein). Nachdem das Feld seinen Zweck erfuellt hat, kann es mit der Anweisung ERASE wieder geloescht werden.

Das folgende Programmbeispiel eroeffnet eine Datei, ermittelt die Anzahl Saetze der Datei und erzeugt ein Feld, dessen Groesse von dieser Anzahl abhaengt:

```
OPEN "BFNR.DAT" AS #1 LEN = 9
SATZANZ = LOF(1)/9                    'Ermitteln Satzanzahl
DIM FOLGE(SATZANZ)
.
.
.
ERASE FOLGE
.
.
.
```

Ist die Groesse eines Feldes grundsaetzlich festgelegt, sollte ein statisches Feld benutzt werden. Ergibt sich die Groesse dagegen erst mit Abarbeitung des Programms, dann ist es sinnvoll, ein dynamisches Feld zu verwenden.

Die Verwendung von dynamischen Feldern kann jedoch auch Probleme bringen, denn hier wird waehrend der Laufzeit zusaetzlicher Speicherplatz angefordert, der eventuell nicht vorhanden ist.

Deshalb sollte vor Erzeugen eines dynamischen Feldes mit der Funktion FRE(-1) die Menge des noch verfuegbaren Speicherplatzes ueberprueft werden.

Compiler-Befehle fuer das Erzeugen von Feldern

Mit den Compiler-Befehlen $STATIC und $DYNAMIC kann kontrolliert werden, welche Art von Feldern der Compiler erzeugt. Die Standardvorgabe ist $STATIC. Im Normalfall wird ein Programm nur einen dieser Compiler-Befehle enthalten.

Diese beiden Compiler-Befehle koennen auch als zusaetzliche Parameter innerhalb einer DIM-Anweisung verwendet werden und gelten dann nur fuer diese eine DIMensionierung.

Einige Beispiele:

```
$STATIC                 'statistisches Feld
DIM a(40),b(20,20)      'a und b sind statisch
DIM DYNAMIC c(10)       'setzt STATIC zeitweilig ausser
                         Kraft

DIM d(x)                'eine Variable: grundsaetzlich
                         dynamisch
DIM STATIC e(x)         'ergibt einen Fehler
$DYNAMIC
DIM f(20)               'trotz einer Konstanten dynamisch
```

Ein dynamisch erzeugtes Feld wird durch den Befehl ERASE komplett geloescht. Der dazugehoerige Speicherplatz steht danach fuer andere Zwecke zur Verfuegung. Die Anwendung von ERASE auf ein statisches Feld loescht seinen gesamten Inhalt (numerische Elemente auf 0, Zeichenketten-Elemente auf die Nullzeichenkette), gibt aber den dazugehoerigen Speicherplatz nicht wieder frei.

3.5. Ausdruecke

Ein **Ausdruck** besteht aus einer Folge von Operatoren und Operanden. Je nachdem, ob die Operanden numerischer Art sind oder aus Zeichenketten bestehen, spricht man von numerischen oder von Zeichenkettenausdruecken.

Ein **Zeichenketten-Ausdruck** besteht aus einer Folge von Zeichenkettenkonstanten, Zeichenkettenvariablen, Zeichenkettenfunktionen und (optional) dem Operator +, der zwei Zeichenketten miteinander verbindet. Das Ergebnis eines Zeichenketten-Ausdrucks ist in jedem Fall wieder eine Zeichenkette - also eine Folge von ASCII-Zeichen, deren Gesamtlaenge bekannt ist.

Beispiel:

```
Z$
"ROBOTRON"
Z$ + x$
```

Ein **numerischer Ausdruck** besteht aus einer Folge von numerischen Konstanten, Variablen und Funktionen, die (optional) durch numerische Operatoren voneinander getrennt sind. Das Ergebnis eines numerischen Ausdrucks ist grundsaetzlich wieder ein numerischer Wert, der (je nach Art des Ausdrucks) eines der vier numerischen Formate von TBASIC annimmt (Integer, Langinteger, einfach oder doppeltgenau Real).

Operatoren lassen sich - abhaengig vom erzeugten Ergebnis - in drei Klassen einteilen:

Arithmetisch, Vergleich und **Logisch.**

Arithmetische Operatoren

Die arithmetischen Operatoren in absteigender Reihenfolge sind:

Operator	Operation	Beispiel
^	Potenzierung	x ^ y
-	Negation	-x
*,/	Multiplikation, Division	x * y , x / y
\	Integer-Division	x \ y
MOD	Modulo	x MOD y
+,-	Addition, Subtraktion	x + y , x - y

Die Integer-Division, dargestellt durch den umgekehrten Schraegstrich (\), rundet zunaechst alle Operanden in Integerwerte und erzeugt danach einen Quotienten, dessen Nachkommastellen abgeschnitten werden:

```
z.B.: 7 \ 3 ergibt als Ergebnis 2
      5 \ 6 ergibt als Ergebnis 0
```

Der bei einer Integer-Division verbleibende Rest kann ueber den Operator MOD (Modulo) bestimmt werden.
MOD fuehrt ebenfalls eine Integer-Division aus, liefert aber den Restwert zurueck:

```
z.B.: 7 MOD 3  ergibt als Ergebnis 1
      5 MOD 6  ergibt als Ergebnis 5
```

Vergleichs-Operatoren

Ueber Operatoren dieses Typs koennen zwei numerische Werte oder zwei Zeichenketten miteinander verglichen werden. Beide Operanden muessen denselben Typ haben. Ein Vergleichs-Operator liefert ein boolsches Ergebnis (TRUE oder FALSE), das im Integer-Format dargestellt wird: **TRUE** entspricht dem Wert -1, **FALSE** entspricht dem Wert 0. Das Ergebnis eines Vergleichs kann als numerischer Wert ausgegeben, in einen numerischen Ausdruck einbezogen oder als Wahrheitswert in einer Anweisung IF / THEN verwendet werden.

Als Vergleichsoperatoren stehen zur Verfuegung :

Operator	Vergleich	Beispiel
=	Gleichheit	x = y
<>	Ungleichheit	x <> y
<	Kleiner als	x < y
>	Groesser als	x > y
<=	kleiner / gleich	x <= y
>=	groesser / gleich	x >= y

Bei einer Kombination von Vergleichs- und arithmetischen Operatoren innerhalb eines Ausdrucks werden die arithmetischen Operationen grundsaetzlich zuerst ausgefuehrt.

Logische Operatoren

Operatoren dieser Art verknuepfen zwei Integerwerte Bit fuer Bit miteinander. Normalerweise handelt es sich bei den Operanden um Wahrheitswerte - zusammen mit Vergleichs-Operatoren lassen sich dadurch komplexe Pruefungen aufbauen.

Es stehen folgende logische Operatoren zur Verfuegung:

NOT
AND
OR
XOR
EQV
IMP

In der folgenden Tabelle werden die Ergebnisse von logischen Ausdruecken dargestellt (T - TRUE, F - FALSE) :

x	y	NOT x	x AND y	x OR y	x XOR y	x EQV y	x IMP y
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	F	T
F	F	T	F	F	F	T	T

Einige Beispiele dazu:

```
 0 AND  0                ergibt False
-1 AND -1                ergibt True
 0 OR  -1                ergibt True
 5>6 OR  6<7             ergibt True
-1 XOR  -1               ergibt False
 5>6 XOR  6<7            ergibt True
-1 EQV -1                ergibt True
 5>6 EQV 6<7             ergibt False
-1 IMP -1                ergibt True
```

Zu beachten ist, dass logische Operatoren intern immer mit Integerwerten arbeiten. Liegen die Operanden ausserhalb des Integerbereiches, findet ein Ueberlauf statt.

```
z.B.: x = 40000
      IF x OR y THEN...
```

Bit-Manipulationen

Mit logischen Operatoren lassen sich nicht nur komplexe Pruefungen konstruieren. Aufgrund der bitweisen Verknuepfung der Operanden sind ueber AND, OR und XOR direkte Manipulationen einzelner Bits moeglich.

Mit **AND** lassen sich einzelne Bits eines Integerwertes gezielt **loeschen**, ohne dabei die anderen Bits dieses Wertes zu veraendern. Das folgende Beispiel loescht das hoechstwertige Bit des Wertes &HB300, indem dieser Wert mit &H7FFF maskiert wird:

```
     1011 0011 0000 0000          Wert
AND  0111 1111 1111 1111          Maske
 =   0011 0011 0000 0000          Ergebnis
```

Innerhalb des Maskenwertes sind nur die Bits geloescht, die auch im Ergebnis zurueckgesetzt werden sollen, alle anderen Bits der Maske sind gesetzt und lassen deshalb den Originalwert unveraendert.

Eine Operation mit **OR setzt** einzelne Bits. Innerhalb des Maskenwertes sind nur die Bits gesetzt, die auch im Ergebnis gesetzt werden sollen. Alle anderen Bits sind geloescht und lassen deshalb den Originalwert unveraendert. Das folgende Beispiel setzt die hoechstwertigsten Bits des Wertes &H1400:

```
     0001 0100 0000 0000          Wert
 OR  1100 0000 0000 0000          Maske
 =   1101 0100 0000 0000          Ergebnis
```

Eine Operation mit **XOR komplementiert** einzelne Bits. Innerhalb des Maskenwertes sind nur die Bits gesetzt, die auch im Ergebnis ihren Zustand veraendern sollen. Alle anderen Bits sind geloescht und lassen deshalb den Originalwert unveraendert. Das folgende Beispiel invertiert die beiden hoechstwertigen Bits des Wertes &HB300:

```
     1011 0011 0000 0000          Wert
XOR  1100 0000 0000 0000          Maske
     0111 0011 0000 0000          Ergebnis
```

Bit-Manipulationen mit XOR werden oft innerhalb von Grafiken verwendet. Mit ihnen laesst sich ein Objekt sehr einfach vor einem Hintergrund bewegen. Eine erste XOR-Verknuepfung des Objekts mit dem Hintergrund laesst es erscheinen, eine Wiederholung dieser Operation laesst es wieder verschwinden, ohne dass dabei der Hintergrund zerstoert wird.

Reihenfolge der Abarbeitung numerischer Ausdruecke

Bei der Formulierung eines numerischen Ausdrucks muss darauf geachtet werden, dass bestimmte Operationen **Vorrang** vor anderen haben. Die folgende Tabelle gibt die Rangfolge der Operatoren wieder:

- Potenzierung (^)
- Minus (-)
- Multiplikation , Division (* und /)
- Integer-Division (\)
- Modulo (MOD)
- Addition , Subtraktion (+ und -)
- Vergleichs-Operatoren (<, <=, =, >=, >, <>)
- NOT
- AND
- OR, XOR
- EQV
- IMP

Bei der Auswertung zusammengesetzter Ausdruecke, deren Operatoren dieselbe Rangfolge haben, bearbeitet TBASIC den Ausdruck von links nach rechts.

Von Klammern umschlossene Operationen werden grundsaetzlich als erstes ausgefuehrt (wobei die Auswertung mit der innersten Klammerebene beginnt). Bei zusammengesetzten Ausdruecken innerhalb einer Klammer gilt wiederum die oben gezeigte Rangfolge der Operatoren.

Vergleichs-Operatoren und Zeichenketten

In TBASIC lassen sich Vergleichs-Operatoren auch auf Zeichenketten anwenden. Zeichenkettenvariablen und -ausdruecke lassen sich auf Gleichheit und Ungleichheit sowie auf "groesser" und "kleiner" (fuer eine alphabetische Sortierung) pruefen.

Zwei Zeichenketten-Ausdruecke werden nur dann als gleich betrachtet, wenn sowohl ihre Inhalte als auch ihre Laenge uebereinstimmen.

Die alphabetische Anordnung von Zeichenketten geschieht nach zwei Kriterien: den dazugehoerigen ASCII-Codes und der Laenge der Zeichenketten. Das Zeichen A (ASCII-Code 65) ist deshalb "kleiner" als das Zeichen B (ASCII-Code 66). Grossbuchstaben (ASCII-Codes von 65 bis 90) sind aus demselben Grund immer "kleiner" als Kleinbuchstaben (ASCII-Codes 97 bis 122).

a ist also "groesser" als B.

Um Zeichenketten unabhaengig von Gross- und Kleinschreibung alphabetisch zu sortieren, koennen die Funktionen UCASE$ oder LCASE$ benutzt werden.

Die Laenge zweier Zeichenketten-Operanden spielt bei einem Vergleich erst dann eine Rolle, wenn beide bis zum Ende der kuerzeren Zeichenkette exakt gleich sind. In diesem Fall wird die kuerzere Zeichenkette als "kleiner" betrachtet.

Beispiel:

```
a$ = "Auto"
b$ = "Autos"
Die Zeichenkette a$ ist kleiner als b$.
```

3.6. Unterprogramme, Funktionen, Prozeduren

Der Aufbau eines in TBASIC geschriebenen Programms kann durch den Einsatz dieser drei Strukturelemente erheblich vereinfacht werden. Ein **Unterprogramm** besteht aus einer Folge von Anweisungen, denen eine Marke (oder eine Zeilennummer) als Name vorangestellt ist, und wird ueber die Anweisung GOSUB aufgerufen. Eine **Funktion** ist ein aehnliches Konstrukt, sie wird zusammen mit Parametern aufgerufen und liefert ein Ergebnis zurueck. Ein **Prozedur** kann als eigenstaendiges Programm betrachtet werden und wird auch **Subprogramm** genannt - hier koennen Hauptfunktionen eines Programms kodiert werden. Durch die Verwendung von komplexen und / oder mehrfach benutzten Ablaeufen in diese Konstrukte laesst sich ein Programm sehr uebersichtlich gestalten.

Prozeduren und Funktionen in TBASIC ermoeglichen echte Rekursion, die Uebergabe von Parametern sowie den Zugriff auf lokale, statische und globale Variablen.

Prozeduren und Funktionen haben vieles gemeinsam - der einzige wirklich offensichtliche Unterschied liegt darin, dass Funktionen immer ein Ergebnis zurueckliefern und deshalb implizit innerhalb von Ausdruecken aufgerufen werden. Prozeduren liefern dagegen kein direktes Ergebnis zurueck und werden aus diesem Grund ueber den Befehl CALL aufgerufen.

3.6.1. Unterprogramme

Ein Formulieren haeufig benoetigter Programmteile als Unterprogramm ist das traditionelle Strukturelement von BASIC. Ein Unterprogramm wird durch einen Namen (oder eine Zeilennummer) bezeichnet, besteht aus einer Folge von Anweisungen und endet mit der Anweisung RETURN. Um diese Anweisungen auszufuehren, muss innerhalb des Hauptprogramms die Anweisung GOSUB zusammen mit dem Namen des entsprechenden Unterprogramms gegeben werden. Durch das RETURN am Ende des Unterprogramms erfolgt ein Ruecksprung. Die Programmausfuehrung wird mit der Anweisung fortgesetzt, die sich direkt hinter der Aufrufstelle (d.h. dem GOSUB) befindet.

Ein Beispiel dazu:

```
GOSUB SUMME          'Aufruf von Summe
PRINT S
END

SUMME:
    S = 0
    FOR i = 1 TO 100
      S = S + i
    NEXT i
RETURN
```

Hinweis:

Waehrend der Entwichlungsphase eines Programms sollte das Pruefen des Stacks (Menue **Options**, Schalter **Stack test**) auf ON gesetzt sein.

3.6.2. Funktionen

TBASIC kennt zwei Arten von Funktionen:

- Vordefinierte Funktionen wie COS und CHR$, die im Sprachumfang enthalten sind (siehe Kapitel 4).
- Benutzerdefinierte Funktionen, die aus einer oder mehreren Programmzeilen bestehen koennen.

Die syntaktische Definition einer einzeiligen Funktion lautet:

DEF FNNAME[(Parameterliste)] = Ausdruck

Name ist der benutzerdefinierte Name, ueber den die Funktion aufgerufen wird. Dieser Name darf maximal 31 Zeichen lang sein und muss den Regeln fuer **Namen** entsprechen (Beginn mit einem Buchstaben, danach eine beliebige Folge von Buchstaben und Ziffern, keine Umlaute).

Parameterliste ist eine (optionale) Folge von Variablennamen, die voneinander durch Kommas getrennt sind. Sie repraesentieren Daten, die der Funktion beim Aufruf uebergeben werden.

Ausdruck steht fuer die Operation, die die Funktion bei ihrem Aufruf durchfuehrt. Das Ergebnis dieser Operation wird als Wert der Funktion zurueckgeliefert.

Mit mehrzeiligen benutzerdefinierten Funktionen lassen sich in TBASIC Dinge erreichen, die weit ueber die Moeglichkeiten der (einzeiligen) Funktionen des BASIC-Interpreters hinausgehen. Eine derartige Funktion kann sich aus einer beliebigen Anzahl von Zeilen und Anweisungen zusammensetzen und verhaelt sich im Prinzip wie ein Unterprogramm - wenn man von verschiedenen Erweiterungen und der Tatsache absieht, dass sie ebenfalls immer ein Ergebnis zurueckliefert. Die formale Syntaxdefinition sieht folgendermassen aus:

```
DEF FNName[(Parameterliste)]
[Variablen-Deklaration]
   .
   .  Anweisungen
   .
   [EXIT DEF]
   [FNFunktionsname = Ausdruck]
END DEF
```

Name ist der Name der Funktion, ueber den sie aufgerufen wird.
Die dafuer gueltigen Regeln sind dieselben wie bei einzeiligen Funktionen.

Parameterliste ist eine (optionale) Folge von Variablennamen, die voneinander durch Kommas getrennt sind und Werte repraesentieren, die der Funktion bei ihrem Aufruf uebergeben werden.

EXIT DEF hat innerhalb von Funktionen dieselbe Wirkung wie die Anweisung RETURN in einem Unterprogramm: Sie bewirkt einen Ruecksprung zur Aufrufstelle. Im internen Ablauf des Programms bestehen zwischen diesen beiden Befehlen allerdings grosse Unterschiede - RETURN kann deshalb zum Abbruch einer Funktion nicht benutzt werden. Innerhalb einer Funktion muss EXIT DEF nur dann verwendet werden, wenn tatsaechlich Anweisungen uebersprungen werden sollen. Der Ruecksprung wird sonst durch END DEF ausgefuehrt.

Beispiel:

Berechnen der Formel $\frac{x + y}{x * y}$
und schreiben einer mehrzeiligen Funktion!

```
DEF FNFORM(x,y)
LOCAL ERG
     IF x = 0 OR y = 0 THEN FNFORM = -1: EXIT DEF
     ERG = (x + y) / (x * y)
FNFORM = ERG
END DEF
```

Die Funktionsdefinition wird von den reservierten Woertern DEF FN und END DEF umschlossen. Es wird eine lokale Variable ERG definiert. Es wird geprueft, ob x oder y den Wert Null besitzen, da eine Division durch 0 nicht erlaubt ist. Ist dies der Fall, so wird der Wert auf -1 gesetzt und die Funktion vorzeitig ueber EXIT DEF verlassen.

Formale und aktuelle Parameter

Variablen, die als Parameter in einer Funktionsdefinition erscheinen, werden **formale** Parameter genannt. Sie dienen nur als Platzhalter und haben mit Variablen gleichen Namens im Hauptprogramm nicht das geringste zu tun. Das folgende Programmbeispiel demonstriert diesen Unterschied:

```
100 DEF FNFlaeche(x,y) = x * y     'einzeilige Funktion
110 x = 56
120 PRINT x, FNFlaeche(2,3), x
```

Die in den Zeilen 110 und 120 verwendete Variable x hat keinerlei Bezug zu dem formalen Parameter x der Funktionsdefinition in Zeile 100. Beim Ablauf dieses Programms behaelt sie deshalb ihren Wert sowohl vor als auch nach dem Funktionsaufruf.

Die Werte, die einer Funktion zur Laufzeit eines Programms uebergeben werden, nennt man **aktuelle** Parameter. Im zuletzt gezeigten Beispiel wurden der Funktion **Flaeche** die Konstanten 2 und 3 als aktuelle Parameter uebergeben. Wobei es keinen Unterschied macht, ob es sich um Konstanten oder Variablen oder eine Mischung aus beiden handelt:

```
a = 2 : b = 3
PRINT FNFlaeche(a,b), FNFlaeche(2,b), FNFlaeche(2,3)
```

Funktionstypen

Funktionen koennen einen numerischen Wert beliebigen Formates (Integer, Langinteger, einfach und doppeltgenau Real) oder eine Zeichenkette zurueckliefern. Der Typ einer Funktion wird auf dieselbe Weise wie bei einer Variablen festgelegt, naemlich durch eine entsprechende Kennzeichnung des Namens.

```
DEF FNINTEGERZ%(x) = INT(20*sin(x))      'eine Integer-
                                          Funktion
```

Der Versuch, einer numerischen Variablen das Ergebnis einer Zeichenkettenfunktion zuzuweisen (oder umgekehrt), erzeugt den Laufzeitfehler 13 (Typkonflikt).

3.6.3. Prozeduren

Prozeduren bestehen aus einer Folge von Anweisungen, die von den Anweisungen **SUB** und **END SUB** eingeschlossen werden. Die formale Syntax einer Prozedurdefinition ist

```
SUB Name[(Parameterliste)] (INLINE)
[Variablen - Deklaration]
      .
      .  Anweisungen
      .
      [EXIT SUB]
END SUB
```

Name ist der Name der Prozedur, der bis zu 31 Zeichen lang sein kann und innerhalb eines Programms nur einmal vorkommen darf. Ueber ihn wird die Prozedur aufgerufen.

Parameterliste ist eine (optionale) Folge von Variablennamen, die durch Kommas voneinander getrennt werden und Daten repraesentieren, die der Prozedur bei einem Aufruf uebergeben werden.

INLINE bedeutet, dass der Prozedur eine nicht festgelegte Anzahl "typenloser" Parameter uebergeben werden kann, und einzelne Zeilen der Prozedur direkt in Maschinensprache geschrieben sind (siehe Kapitel 5).

Der hauptsaechlichste Unterschied zwischen Funktionen und Prozeduren besteht darin, dass letztere keine Werte zurueckliefern. Prozeduren werden nicht implizit innerhalb von Ausdruecken aufgerufen, haben keinen eigenen Typ und enthalten keine Zuweisung zu ihrem Namen. Sie werden mit dem Befehl CALL aufgerufen und aehneln in diesem Punkt sehr stark einem normalen Unterprogramm, das mit GOSUB gestartet wird.

Das folgende Beispiel deklariert eine Prozedur Namens **Gesamt** und ruft sie anschliessend auf:

Prozedur:

```
SUB Gesamt(a,b,c)
LOCAL SUM
  SUM = a + b + c
  PRINT SUM
END SUB
```

Hauptprogramm:

```
      .
      .
x = 10 : y = 20 : z = 30
CALL Gesamt(x,y,z)
```

Felder als Prozedur - Parameter

Im Gegensatz zu Funktionen koennen Prozeduren nicht nur einfache Variablen, sondern auch komplette Felder als Parameter uebergeben werden. Die Definition der Prozedur muss natuerlich entsprechend ausgelegt sein: die Parameterliste muss eine Feldvariable enthalten. Der Typ dieses formalen Parameters wird genauso wie bei normalen Variablen festgelegt, darauf folgt die **Anzahl der Dimensionen** in Klammern.

Beispiel:

```
SUB MATRIX(M(2))
                    'das Argument von MATRIX ist ein zwei-
                    dimensionales  Feld vom Typ Real  ein-
                    fache Genauigkeit.
```

3.6.4. Verwendung von Parametern in Funktionen und Prozeduren

Zwischen Funktionen und Prozeduren gibt es einige Unterschiede, deren Verstaendnis eine etwas detaillierte Erklaerung der inneren Ablaeufe von TBASIC erforderlich macht.

Unterschiede zwischen Prozeduren und Funktionen:

Wirkung	Funktionen	Prozeduren
Zurueck mit	einem Wert	nichts
Aufruf	durch Ausdruecke	durch Anweisung CALL
Parameteruebergabe	Wert	Adresse und Wert
undeklarierte Variablen	SHARED	STATIC
Felder als Parameter	nein	ja

Beispiel:

Berechnen der Kreisflaeche

```
DEF FNKREIS(radius)
    FNKREIS = radius * radius * 3.14159
END DEF

r = 7.9
FLAECHE = FNKREIS(r)
```

Wird dieses Programm gestartet, erfolgt zuerst die Zuweisung der Konstanten an die Variable r. Anschliessend wird FNKREIS mit dieser Variablen aufgerufen.

Die Uebergabe des Wertes an die Funktion kann auf zwei verschiedene Arten erfolgen:

1. Uebergabe des Wertes 7.9 - Wertuebergabe
2. Uebergabe der Variablen r - Adressuebergabe

Wertuebergabe: Hier werden die Werte der Argumente gelesen und in einen Zwischenspeicher kopiert. Die aufgerufene Routine kann auf diesen Bereich zugreifen, die Werte von dort lesen/veraendern, und wenn sie beendet ist, wird der Zwischenspeicher sofort wieder freigegeben. Die Original-Argumente bleiben dabei unveraendert.

Adressuebergabe: Hier wird ein Zeiger auf die einzelnen Argumente uebergeben (im Beispiel ein Zeiger auf die Speicherzelle, in der der tatsaechliche Wert der Variablen r steht). Die aufgerufene Routine kann mit Hilfe dieser Zeiger die Werte der Variablen lesen und auch veraendern.

Bei einer Wertuebergabe koennen Konstanten und/oder Ausdruecke als Argument benutzt werden. Zur Laufzeit des Programms wird der entsprechende Ausdruck berechnet bzw. der Wert der Konstanten eingesetzt und der Funktion uebergeben.

Damit ist z.B. auch ein Aufruf wie der folgende moeglich:

```
v = FNKREIS(r1 * 2 + 5)
```

Der Wert des Arguments wird berechnet und uebergeben.

Eine Adressuebergabe erlaubt der aufgerufenen Routine die Manipulation der uebergebenen Variablen und kann deshalb fuer Konstanten und Ausdruecke nicht funktionieren (ein Ausdruck wie h * 2 + 4.1 erzeugt nur ein Zwischenergebnis, das keine fixe Adresse im Speicher des Computers hat). Adressuebergaben sind deshalb nur dann moeglich, wenn das Argument eine Variable ist.

Hinweise zur Uebergabe von Parametern:

1. Mit der Wertuebergabe koennen sowohl Variablen als auch Konstanten und Ausdruecke uebergeben werden. Adressuebergaben sind nur fuer Variablen als Argumente moeglich.

   Der Vorteil der Adressuebergabe liegt darin, dass die aufgerufene Routine die entsprechenden Variablen veraendern und somit Ergebnisse (je nach Anzahl der Argumente auch mehr als einen Wert) zurueckliefern kann. Bei der Wertuebergabe wird nur eine Kopie des Arguments uebergeben, deren Veraenderung keinen Einfluss auf den "Originalwert" hat.

2. Das Erzeugen einer Kopie der Argumente benoetigt Zeit und Speicherplatz; umso mehr, je umfangreicher das Argument ist. Aus diesem Grund laesst TBASIC die Uebergabe von Feldern nur ueber ihre Adresse zu (und damit nur bei Prozeduren).

   Das folgende Programm illustriert die Adressuebergabe beim Aufruf einer Prozedur:

```
a =10 : b = 50
CALL SUMME(a, b*b, s)
PRINT a,b,s
END

SUB SUMME (i,j,k)
   k = i + j
   i = i + 1
   j = j + 1
END SUB
```

   Nach dem Aufruf der Prozedur **SUMME** enthaelt s den Wert der Addition von a,b. Die Variable hat einen neuen Wert (Adressuebergabe).
   Die Variable b dagegen hat ihren alten Wert behalten, obwohl der Parameter j ebenfalls veraendert wurde innerhalb der Prozedur SUMME. Da das zweite Argument aber als Ausdruck uebergeben wurde, erfolgte eine Wertuebergabe, und demzufolge ist keine Aenderung von b moeglich.

3. Die Argumente von Funktionen werden als Werte, die von Prozeduren als Werte oder Adressen uebergeben.

- Wird beim Aufruf einer Prozedur der Name einer Variablen direkt als Argument angegeben, erfolgt eine Adressuebergabe.

- Werden beim Aufruf einer Prozedur als Argumente Ausdruecke und/oder Konstanten verwendet, erfolgt eine Wertuebergabe.

- Ist das Veraendern eines Argumentes durch die Prozedur nicht erwuenscht, dann kann die entsprechende Variable als Ausdruck deklariert werden, indem der Variablenname in Klammern gesetzt wird. In diesem Fall erfolgt eine Wertuebergabe.

- Funktionen koennen mit Konstanten, Ausdruecken, Variablen und Feldelementen (wie Variable) als Argumente aufgerufen werden.

- Felder werden grundsaetzlich ueber ihre Adresse uebergeben und sind deshalb als Argumente nur bei Prozeduren moeglich.

3.6.5. Lokale Variable

Im BASIC-Interpreter sind alle Variablen global, d.h. sie gelten innerhalb jedes Programmteils.

TBASIC erlaubt die Verwendung "lokaler" Variablen innerhalb von Prozeduren und Funktionen. Eine als LOCAL deklarierte Variable existiert nur innerhalb der Routine, in der sie erzeugt wurde.

Felder lassen sich ebenfalls lokal erzeugen:

- Definieren einer entsprechenden Feldvariablen
- Verwenden der Anweisung DIM

Beispiel:

```
SUB TEST
LOCAL x, FELD()
DIM DYNAMIC FELD(10)


END SUB
```

Der durch ein lokales Feld belegte Speicherplatz wird beim Verlassen der Prozedur bzw. Funktion automatisch wieder freigegeben.

Lokale Variablen haben folgende Eigenschaften:

1. Sie muessen deklariert werden, bevor sie das erste Mal innerhalb der Prozedur bzw. Funktion benutzt werden.

2. Bei der Deklaration wird der Wert einer numerischen lokalen Variablen auf 0, Zeichenketten auf Leerzeichenkette ("") gesetzt.

3. Lokale Definitionen haben **Vorrang** vor globalen Definitionen. Ist innerhalb einer Prozedur oder Funktion eine lokale Variable definiert, kann von dieser Prozedur oder Funktion aus auf eine globale Variable gleichen Namens nicht mehr zugegriffen werden.

4. Die Werte lokaler Variablen bleiben zwischen zwei Aufrufen nicht erhalten.

3.6.6. Attribute SHARED und STATIC, nichtdeklarierte Variable

SHARED

Dieses Attribut ist das exakte Gegenteil von LOCAL. Um innerhalb einer Funktion oder Prozedur auf eine globale Variable zugreifen zu koennen, muss diese Variable zuvor als SHARED deklariert werden.

STATIC

Variablen mit diesem Attribut liegen in ihren Eigenschaften zwischen LOCAL und SHARED. Wie bei einer Deklaration mit LOCAL erzeugt die Variable keine Ueberschneidungen mit einer globalen Variablen des Programms, die denselben Namen hat. Sie verfuegt ueber einen festen Platz im Speicher und verhaelt sich in diesem Punkt wie eine Variable mit dem Attribut SHARED. Das heisst, mit STATIC deklarierte Variablen sind lokal, behalten ihren Wert aber zwischen einzelnen Aufrufen der Prozedur bzw. Funktion. Variablen dieser Art werden nur beim Start des Programms automatisch zurueckgesetzt (auf den Wert 0 bzw. eine Leerzeichenkette).

Durch eine streng formale Definition der Argumente und lokaler Variablen koennen Prozeduren und Funktionen so ausge-legt sein, dass sie voellig unabhaengig von dem Programm sind, in dem sie benutzt werden. Durch das Konzept der Haupt- und Arbeitsdatei ("Main" und "Work file") von TBASIC zusammen mit dem Compiler-Befehl $INCLUDE ist eine problemlose Uebernahme und Verwaltung eigener Funktions-Bibliotheken moeglich.

Nicht deklarierte Variablen

In fast allen Compilersprachen wie PASCAL, FORTRAN, C usw. reagiert der Compiler mit einer Fehlermeldung, wenn er eine nicht deklarierte Variable entdeckt.

In TBASIC gibt der Compiler in benutzerdefinierten Funktionen nicht deklarierten Variablen das Attribut SHARED, d.h. behandelt sie als global.

In Prozeduren gibt der Compiler nicht deklarierten Variablen das Attribut STATIC, d.h. behandelt sie als lokale Werte mit festem Speicherplatz.

3.6.7. Rekursion

TBASIC unterstuetzt rekursive Definition von Prozeduren und Funktionen, d.h. Ablaeufe, in denen sich eine Routine selbst aufruft.

Beispiel:

Funktion Fakultaet

```
DEF FNFAK#(n%)
   IF n%>1 AND n%<170  THEN
      FNFAK# = n% * FNFAK#(n%-1)
      ELSEIF n% = 0  OR  n% = 1  THEN
          FNFAK# = 1
      ELSE
          FNFAK# = -1
      END IF
END DEF
```

Vor der Benutzung eines rekursiv formulierten Algorithmus sollte der Compiler-Befehl $STACK benutzt werden, um genuegend Platz fuer die Zwischenspeicherung zu schaffen. Ein rekursiver Aufruf belegt im Durchschnitt 12 Byte im Stack (variiert in Abhaengigkeit von der Anzahl der uebergebenen Argumente).
Ueber die Funktion FRE(-2) laesst sich der noch zur Verfuegung stehende Platz bestimmen.

3.7. Dateiarbeit

TBASIC kennt drei unterschiedliche Arten von Dateien:

Sequentielle Dateien, Direktzugriffsdateien und Binaerdateien.

Die Bezeichnung fuer Dateien, die mit TBASIC erzeugt werden, muss den Regeln von DCP entsprechen. Fuer Dateibezeichnungen sind hier ebenfalls die Umlaute und ss zugelassen. Im Gegensatz zu Marken muss das erste Zeichen nicht unbedingt ein Buchstabe sein.

Die Angabe eines Suchpfades ist ebenfalls moeglich.

Sequentielle Dateien

Eine sequentiell organisierte Datei besteht aus Folgen von ASCII-Zeichen. Jede einzelne Folge kann eine (fast) beliebige Laenge haben und wird von der naechsten durch Wagenruecklauf / Zeilenschaltung abgetrennt.
Einer der Hauptgruende fuer die Benutzung von Dateien dieses Types ist ihre fast uneingeschraenkte Portabilitaet. Sie koennen von Text-Editoren gelesen, in Datenbanken eingebunden oder ueber die seriellen Ports zu anderen Geraeten gesendet werden.

Direktzugriffsdateien

Dateien dieser Art bestehen aus einer Anzahl von Saetzen, auf die in beliebiger Folge zugegriffen werden kann.
Ein Nachteil dieser Organisationsform ist u.a. die geringe Portabilitaet von Direktzugriffsdateien: So ist es z.B. nicht ohne weiteres moeglich, den Inhalt einer Direktzugriffsdatei mit einem Texteditor zu bearbeiten, oder auf einen anderen Computer oder eine andere Programmiersprache zu uebertragen.

TBASIC benutzt zur Speicherung von Realwerten einfacher und doppelter Genauigkeit das international genormte IEEE-Format, im Gegensatz zum Basic-Interpreter BASI.

Um Dateien in beiden Systemen verarbeiten zu koennen, besitzt TBASIC spezielle Konvertierungsfunktionen:

- CVMS und CVMD verwandeln das Format von BASI erzeugter Realzahlen in das IEEE-Format
- MKMS$ und MKMD$ erlauben das Schreiben von Realzahlen im von BASI geforderten Format.

Integer-Werte und Zeichenketten sind in beiden Formaten gleich definiert.

Ein charakteristisches Merkmal von Direktzugriffsdateien ist, dass alle Datensaetze dieselbe Laenge haben muessen.

Binaerdateien

Die Moeglichkeit, eine beliebige Datei als binaer zu erklaeren, stellt eine Erweiterung dar.
TBASIC behandelt eine derartige Datei als voellig willkuerliche (unstrukturierte) Folge von Bytewerten. Die Bearbeitung einer Datei reduziert sich darauf, wieviele Bytes wo in dieser Datei gelesen oder geschrieben werden sollen.

Jede Datei, die von TBASIC als BINARY eroeffnet wird, bekommt einen Positionszeiger zugeordnet, der die momentane Position innerhalb dieser Datei repraesentiert. Die Funktion LOC liefert den Wert dieses Positionszeigers zurueck, mit der Anweisung **SEEK** laesst sich eine bestimmte Position setzen.

Ein- und Ausgabe mit Peripheriegeraeten

In TBASIC koennen Peripheriegeraete des Computers (z.B. Tastatur, Bildschirm, Drucker usw.) als sequentielle Dateien behandelt werden.
Jedes Geraet hat einen speziellen, reservierten Namen, der mit einem Doppelpunkt endet:

KYBD:	Tastatur, nur Eingabedatei
SCRN:	Bildschirm, nur Ausgabedatei
LPT1:...LPT3:	Drucker, nur Ausgabedatei
COM1:, COM2:	serielle Ports, Ein- und Ausgabedatei.

3.8. Sprachunterschiede zwischen TBASIC und BASI(DCPX)

3.8.1. Nicht unterstuetzte Kommandos und Anweisungen

- Kommandos

Da TBASIC ueber einen bildschirmorientierten Editor verfuegt und zusaetzliche Funktionen ueber Menues abgewickelt werden, sind die entsprechenden Kommandos des BASIC-Interpreters BASI (Quelltext-Korrektur, Laden und Speichern usw.) ueberfluessig und deshalb nicht implementiert. Das sind folgende Kommandos, die normalerweise nur im "Direktmodus" eingegeben werden koennen:

AUTO	**LOAD**
DELETE	**NEW**
EDIT	**RENUM**
LIST	**SAVE**
LLIST	

Das Kommando **EDIT** dient zur Korrektur einzelner Zeilen und wird durch den Editor von TBASIC mehr als vollstaendig ersetzt.

DELETE loescht eine Programmzeile. Dies laesst sich innerhalb des Editors mit CONTROL-Y bzw. den Kommandos zum Markieren und Loeschen eines Blocks wesentlich einfacher ausfuehren.

Da TBASIC ohne Zeilennummern auskommt, sind **AUTO** und **RENUM** ebenfalls ueberfluessig bzw. ohne Funktion.

LIST zum Auflisten geschriebener Programmzeilen ist ebenfalls nicht erforderlich, LIST wird innerhalb des Editors durch die Kursortasten ersetzt.

Das Laden, Speichern und Loeschen von Programmen (**LOAD**, **SAVE** und **NEW**) geschieht in TBASIC ueber eine entsprechende Auswahl im Menue **Options** bzw. durch den Druck auf eine Funktionstaste innerhalb des Editors.

Das einzige Kommando, das wirklich fehlt, ist **LLIST**. Um einen Quelltext auf einen angeschlossenen Drucker auszugeben, muss TBASIC verlassen und der Quelltext mit einem DCP-Kommando zum Drucker kopiert werden.

```
C>COPY PROG.BAS LPT1
```

- Anweisungen

- **CONT**

TBASIC erlaubt nicht die Fortsetzung eines zuvor mit <CTRL-BREAK> oder der Anweisung STOP unterbrochenen Programms. Beide haben die gleiche Wirkung wie die Anweisung END.

- MERGE
Das Vermischen eines vom Compiler erzeugten Maschinenprogramms mit einem weiteren Quelltext, der aus ASCII-Zeichen besteht, ist nicht moeglich.

- USR
Die Anweisung CALL von TBASIC stellt eine starke Erweiterung von USR dar.

3.8.2. Kommandos und Anweisungen mit unterschiedlicher Wirkung

- RUN
Ein in TBASIC geschriebenes Programm kann dieses Kommando benutzen, um sich entweder selbst neu zu starten oder um ein anderes Programm in den Speicher zu laden und es auszufuehren. Die Moeglichkeit, ein Programm mit RUN ab einer bestimmten Zeile zu starten, ist nicht gegeben.

- CALL
Diese Anweisung hat in TBASIC drei unterschiedliche Formen:

```
CALL Prozedurname
CALL ABSOLUTE
CALL INTERRUPT
```

CALL ABSOLUTE und CALL INTERRUPT verwenden die Anweisung bzw. die Funktion **REG** zur Uebergabe von Daten (in BASI wird dazu eine Argumentliste benutzt, deren Variablenadressen auf den Stack gebracht werden).

- CHAIN
Programmteile, die mit CHAIN gestartet werden sollen, muessen als "TBC"-Datei compiliert werden (Schalter **Compile to Chain** im Menue **Options**). Sowohl das aufrufende als auch das neu geladene Programm benoetigen eine Deklaration uebergebener Variablen mit COMMON, wobei Typ und Reihenfolge in beiden Programmen uebereinstimmen muessen (unterschiedliche Namen sind problemlos - zumindest fuer den Compiler).

- DEFTyp, TRON, TROFF und OPTION BASE
Die Wirkung dieser vier Anweisungen unterscheidet sich in TBASIC in einem Detail: Der Compiler beruecksichtigt die physikalische Position innerhalb des Quelltextes - und nicht den spaeteren Ausfuehrungsweg zur Laufzeit des Programms. Das folgende Beispiel verdeutlicht den Unterschied anhand der Anweisung DEFTyp.

```
10 GOTO 30
20 DEFINT A
30 A = 5.3
40 PRINT A
```

Der BASIC-Interpreter wuerde bei der Ausfuehrung dieses Programms die Zeile 20 ueberspringen und folglich die Anweisung DEFINT A nicht zur Kenntnis nehmen. Aus diesem Grund bekommt die Variable A in Zeile 30 den Standardtyp einfacher Genauigkeit zugewiesen, der PRINT-Befehl in Zeile 40 gibt den Wert 5.3 aus. Der Compiler von TBASIC verarbeitet dagegen die DEFINT-Anweisung im Quelltext: Da sie sich vor der Definition der Variablen A befindet, bekommt A den Typ Integer, der PRINT-Befehl in der vierten Zeile des Programms gibt den Wert 5 aus.

- DRAW und PLAY

Diese beiden Befehle interpretieren Befehls-Zeichenketten und ermoeglichen das Ausfuehren von Noten- bzw, Zeichen-Sequenzen. Bei BASI koennen die Befehls-Zeichenketten Namen von numerischen und Zeichenketten-Variablen enthalten, wobei DRAW bzw. PLAY mit dem Inhalt der entsprechenden Variablen arbeiten. Eine Befehlsfolge wie

```
X = 50
PLAY "T=X;C"
```

setzt in BASI das Tempo auf den Wert der Variablen X (d.h. auf 50) und spielt dann die Note C. TBASIC benutzt Variablenamen dagegen nur waehrend des Compilierens und spart den entsprechenden Speicherplatz waehrend der Laufzeit des Programms ein - aus diesem Grund muss die Suche von PLAY nach der Variablen mit dem Namen X erfolglos bleiben. Entsprechende Befehle in existierenden Programmen muessen deshalb folgendermassen umgeschrieben werden:

```
X = 50
PLAY "T=" + VARPTR$(X) + "C"
```

Weitere Details sind unter 4. "Sprachbeschreibung" nachzulesen.

3.8.3. Wirkungsweise der Funktionen

- Definitionen

Funktionen koennen in TBASIC sowohl einzeilig als auch mehrzeilig definiert werden. Die letzte Form ist bei BASI nicht moeglich. Zusammen mit dem Prozedur-Konzept, das bei BASI nicht vorhanden ist, ergeben sich wesentlich verbesserte Strukturierungsmoeglichkeiten.

- Genauigkeit

Die Exponential- und trigonometrischen Funktionen von TBASIC (EXP, LOG, ..., SIN,TAN usw.) liefern Werte doppelter Genauigkeit.

3.8.4. Erweiterung der Speicherausnutzung

Mit TBASIC kann der Speicher des Computers vollstaendig genutzt werden, die Begrenzung auf 64 KByte Programmlaenge bzw. Datengroesse entfaellt. Programme koennen eine beliebige Laenge haben, fuer "einfache" Variablen stehen 64 KByte, fuer Zeichenketten-Variablen weitere 64 KByte und fuer Feldvariablen jeweils 64 KByte Speicherplatz zur Verfuegung. Voraussetzung dafuer ist natuerlich, dass der Computer ueber die RAM-Speichererweiterung verfuegt.

3.8.5. Erweiterung des Sprachumfangs

TBASIC ist gegenueber dem BASIC-Interpreter BASI um eine Vielzahl neuer Anweisungen und Befehle sowie um zusaetzliche Optionen existierender Befehle erweitert worden. Die folgende Tabelle gibt nur die wichtigsten wieder:

INCR/DECR	Alternative zur Erhoehung/Erniedrigung von Variablen
BINARY	ermoeglicht den Dateizugriff auf Byte-Ebene
CVMS,CVMD, MKMS$,MKMD$	Uebersetzungsfunktionen fuer BASI-Dateien
DEFLNG,MKL$,CVL	Operationen mit dem neuen Datentyp Langinteger
DELAY	Verzoegerung um n Sekunden
DO/LOOP	flexibles Schleifenkonstrukt nach dem ANSI-Standard
EXIT	verlaesst FOR/NEXT-, WHILE/WEND- und DO/LOOP-Schleifen, blockstrukturierte IF's und Prozeduren/Funktionen
GET$	liest Bytes von BINARY-Dateien
IF-Blockstruktur	mehrere IF-Tests in Folge
INLINE	direktes Einfuegen von Maschinencode
MEMSET	reserviert freien Speicherplatz oberhalb des Programmes
MTIMER	Zugriff auf den Mikrosekunden-Zaehler
PUT$	schreibt Bytes in BINARY-Dateien
REG	setzt/liest Prozessor-Register
SEEK	setzt die Position in BINARY-Dateien
SELECT	erlaubt Mehrweg-Verzweigungen
CEIL,FIX	Realzahlen --> Integerwerte
ERADR	liefert die Adresse des letzten Fehlers
EXP10,EXP2	liefert Potenzen zur Basis 10 bzw. 2
INSTAT	prueft den Status der Tastatur
VARSEG	liefert die Segmentadresse einer Variablen bzw. eines Feldelements
BIN$	liefert das binaere Aequivalent eines Integers
COMMAND$	liefert Zusatzparameter der Kommandozeile
LCASE$	uebersetzt auf Kleinbuchstaben
LOCAL/SHARED/ STATIC	deklarieren Prozedur- und Funktionsvariablen
SUB/END SUB	definieren einer Prozedur

Weitere Details sind unter 4. "Sprachbeschreibung" nachzulesen.

3.8.6. Laenge der Zeichenketten

Die maximale Laenge einer Zeichenkette in TBASIC betraegt nicht 255 Zeichen wie bei BASI, sondern 32767 Zeichen. Die Verwaltung von Zeichenketten wurde radikal veraendert. Damit entfaellt die Notwendigkeit, aus Speicherplatzgruenden periodisch die FRE-Funktion zu nutzen.

3.8.7 Umwandlung von BASI-Programmen in TBASIC-Programme

Um BASI-Programme unter TBASIC zu verarbeiten, sind BASI-programme mit dem BASIC-Interpreter BASI als ASCII-Datei (Kommando "SAVE", Option "A") zu speichern.
ASCII-Dateien koennen vom TBASIC-Editor direkt gelesen werden. Falls des Programm laenger als 60 KByte sein sollte, muesste dies auf eine entsprechende Anzahl von Dateien verteilt werden.

4. Sprachbeschreibung

Die Sprachbeschreibung von TBASIC setzt Grundkenntnisse in der Programmiersprache BASIC voraus und haelt sich an die Gliederung der Sprachbeschreibung fuer den BASIC-Interpreter BASI (DCPX).

So keine Abweichungen in den Kommandos, Anweisungen und Funktionen vorhanden sind, werden diese in der TBASIC-Sprachbeschreibung ohne Bemerkungen nur aufgefuehrt. Zur Wirkungsweise ist, wenn erforderlich, in der Bedienungsanleitung und Sprachbeschreibung des BASIC-Interpreters BASI nachzulesen.

Abweichungen und Ergaenzungen zu den einzelnen Befehlen einschliesslich neuer Befehle und neuer Gliederungspunkte sind der Sprachbeschreibung fuer TBASIC zu entnehmen.

4.1. Kommandos

4.1.1. Programmausfuehrungskommandos

- RUN

Start eines Programms.

Syntax: **RUN [Dateiname]**

Bemerkungen:

RUN ohne zusaetzliche Angaben startet das momentan im Speicher befindliche Programm. Damit wird gleichzeitig ein CLEAR ausgefuehrt:

- Saemtliche dynamischen Felder werden vollstaendig geloescht, die Elemente statischer Felder und statische Variablen werden auf den Wert 0 bzw. auf eine Nullzeichenkette gesetzt.

- Der DATA-Zeiger zeigt danach auf das erste DATA-Element, Hintergrundprozesse wie PLAY und mit ON...GOSUB und ON ERROR gesetzte "Abfangreaktionen" werden inaktiv.

Der Parameter **Dateiname** kann wahlweise angegeben werden.

Wenn **Dateiname** ohne Dateityp angegeben wird, dann haengt TBASIC automatisch ".TBC" an. Wird ein Dateityp angegeben, bleibt er unveraendert.

RUN zusammen mit der Angabe eines Dateinamens bewirkt, dass TBASIC die bezeichnete Datei in den Speicher laedt und diese startet. Die Datei muss als ".TBC" erzeugt worden sein, d.h. vor dem Compilieren muss der Schalter **Compile to** im Menue **Options** auf **CHAIN** gesetzt werden.

Im Gegensatz zur Anweisung CHAIN findet keine Uebergabe von Daten zum neu geladenen Programm statt.

Unterschiede:
Das Kommando RUN kann nicht mit einer Zeilennummer oder der Angabe "R" gestartet werden.

Beispiel:

Die beiden folgenden Beispielprogramme demonstrieren die unterschiedlichen Moeglichkeiten des Kommandos RUN. Das erste Beispiel muss als .EXE-Datei, das zweite als .TBC-Datei compiliert werden. Um diese Programme auszuprobieren, muss TBASIC verlassen werden. RUN und CHAIN sind nur von der DCP-Kommandoebene aus moeglich.

Programm 1 - muss als .EXE-Datei compiliert werden

```
CLS
PRINT "Moment..";
DELAY 2
' ein Hintergrundprozess
ON TIMER(1) GOSUB Uhrzeit
TIMER ON
LOCATE 4,1
PRINT "Neustarten [1]"
PRINT "Naechstes Programm [2]"

DO
  LOCATE 7,1
  INPUT "Auswahl: ", Wahl$
LOOP UNTIL (Wahl$="1" OR Wahl$="2")

IF Wahl$="1" THEN
  RUN 'Neustart'
ELSE
  RUN "TEIL2" 'Start TEIL2.TBC
END IF
END  'Programmende - wird nie erreicht

Uhrzeit:
  H%=POS(X): V%=CSRLIN
  LOCATE 1,1: PRINT TIME$
  LOCATE V%,H%
RETURN
```

Programm 2 - muss unter dem Namen TEIL2 als .TBC compiliert werden

```
CLS
PRINT "Das ist Teil 2"
END
```

- CLEAR

Der Speicherbereich fuer Variable wird geloescht.

Systax: **CLEAR**

Bemerkungen:

Das Kommando wird ohne Parameter eingegeben und hat folgende Wirkungen:

- Einfache numerische Variablen bekommen den Wert 0, einfache Zeichenkettenvariablen eine leere Zeichenkette ("") zugewiesen.
  Dasselbe gilt fuer die Elemente von statisch DIMensionierten Feldern.

- Dynamisch erzeugte Felder werden durch CLEAR ebenfalls zurueckgesetzt.

Unterschiede:
TBASIC unterstuetzt nicht ein dynamisches Setzen der Stack- und Datensegmente, aus diesem Grund hat der Befehl CLEAR keine Parameter.
Die Groessen des Stack- und Datenbereiches werden von BASI zur Laufzeit gesetzt, von TBASIC dagegen waehrend des Compilierens.

- TRON/TROFF

Ein schrittweises Verfolgen des Programmablaufes zur Fehlersuche ist damit moeglich.

Syntax: **TRON**
TROFF

Bemerkungen:

Mit dem Compiler-Befehl $TRACE (oder dem entsprechenden Schalter in **Options**) kann festgelegt werden, dass der gesamte Ablauf des Programms verfolgt wird.

TRON und TROFF erlauben das Verfolgen einzelner Programmteile.

Beide Kommandos sind ausschliesslich zur Fehlersuche gedacht. Wenn das Programm innerhalb der Umgebung von TBASIC ausgefuehrt wird, erscheinen TRACE-Ausgaben innerhalb des Fensters **Trace**, sind also von den restlichen Ein- uns Ausgaben des Programms getrennt. Beim Start eines Programms von DCP aus existieren keine Fenster - TRACE-Ausgaben erscheinen in diesem Fall vermischt mit den Ausgaben des Programms auf dem Bildschirm.

Nach einem TRON gibt TBASIC Zeilennummern, die Namen von Labels sowie die Namen aufgerufener Prozeduren und Funktionen im Fenster **Trace** aus - solange bis ein TROFF-Befehl erreicht wird.

Nachdem im Fenster **Trace** die erste Ausgabe erschienen ist, haelt TBASIC die Programmausfuehrung an und wartet auf einen der beiden folgenden Tastendruck-Befehle:

- <ALT-F10> ("Einzelschrittmodus"): Nach Eingabe dieser Tastenkombination wird die Programmausfuehrung bis zur naechsten TRACE-Ausgabe fortgesetzt, danach wartet TBASIC auf eine erneute Eingabe von <ALT-F10>.

- <ALT-F9>: Mit dieser Tastenkombination laesst sich zwischen "Einzelschritten" und "kontinuierlich" umschalten. Eine erste Eingabe von <ALT-F9> bewirkt, dass TBASIC nach weiteren **TRACE**-Ausgaben nicht mehr auf einen Tastendruck wartet, die naechste Eingabe von <ALT-F9> schaltet wieder auf "Einzelschritt" um usw.

Unterschiede:
Im Gegensatz zu BASI bestimmt TBASIC anhand der physikalischen Position der Befehle **TRON** und **TROFF** im Quelltext, welche Programmteile verfolgt werden sollen und welche nicht. Dadurch kann sich ein sehr unterschiedliches Verhalten ergeben. Ein Beispiel:

```
10 GOTO 30
20 TRON
30 X = Y + Z
40 TROFF
50 END
```

BASI ignoriert den **TRON**-Befehl in Zeile **20** komplett. Der Compiler von TBASIC kennt den Verlauf eines Programms waehrend der Uebersetzung nicht: Er liest den **TRON**-Befehl in der zweiten Zeile und erzeugt fuer die folgenden Programmschritte zusaetzlichen **TRACE**-Code. Nach den Start dieses Programms erfolgt folgende Ausgabe:

```
[30] [40]
```

Im folgenden Beispiel wuerde ein BASIC-Interpreter bei **TRACE** saemtliche definierten Zeilennummern ausgeben - von TBASIC erhaelt man dagegen nur die Ausgaben [30] und [40].
Der Grund: Das Unterprogramm liegt physikalisch **hinter** dem Befehl **TROFF**.

```
 10 TRON
 30 GOSUB 100
 40 PRINT "Wieder zurueck"
 50 TROFF
 60 END

100 PRINT "Nicht verfolgtes Unterprogramm"
110 RETURN
```

- SYSTEM

Ein Programm wird beendet.

Syntax: **SYSTEM**

Bemerkungen:

SYSTEM beendet ein Programm in derselben Weise wie **END**. Saemtliche offenen Dateien werden geschlossen, danach folgt ein Ruecksprung zur Kommandoebene von DCP bzw. zur Programmierumgebung von TBASIC, je nachdem, von wo aus das Programm gestartet wurde.

Unterschiede:
In BASI wird dieses Kommando benutzt, um nicht nur das Programm, sondern auch den BASIC-Interpreter zu verlassen.

Nachdem dieses Kommando rein aus Gruenden der Kompatibilitaet zu existierenden Programmen definiert wurde, sollte es in neuen Programmen durch **END** ersetzt werden.

4.1.2. Diskettenbezogene Kommandos

- RESET

Saemtliche evtl. noch nicht aufgezeichneten Dateipufferinhalte werden aufgezeichnet und danach alle Dateien geschlossen.

Syntax: **RESET**

Bemerkungen:

RESET hat genau die gleiche Wirkung wie die Anweisung **CLOSE** ohne Angabe von Dateinummern.

- NAME

Eine Datei erhaelt einen neuen Namen.

Syntax: **NAME Dateiname1 AS Dateiname2**

Bemerkungen:
keine

- FILES

Anzeige des Bibliotheksverzeichnisses (Directory).

Syntax: **FILES [Dateiname]**

Bemerkungen:
keine

- KILL

Eine Diskettendatei wird geloescht.

Syntax: **KILL Dateiname**

Bemerkungen:
keine

4.2. BASIC-Grundanweisungen

4.2.1. Kommentaranweisung

- REM

Einfuegen erklaerender Bemerkungen (Kommentare) in ein Programm.

Syntax: **REM Bemerkung**

Bemerkungen:

Bemerkung ist ein Kommentar beliebiger Laenge, der allerdings nicht ueber das Ende der Zeile hinausgehen darf.

Im Gegensatz zu BASI werden bei TBASIC Kommentare nur im Quelltext erforderlich. Sowohl in Bezug auf die Laenge als auch auf die Ausfuehrungsgeschwindigkeit des Objektcodes haben sie keinen Einfluss.

4.2.2. Datum, Uhrzeit, Lautsprecher

- DATE$

Setzt und liest das computerinterne Datum.

Syntax: Als Variable: **v$ = DATE$**
Als Anweisung: **DATE$ = x$**

Bemerkungen:
keine

- TIME$

Enthaelt die Uhrzeit des Systems, die sowohl gesetzt oder gelesen werden kann.

Syntax: **TIME$ = x$** (Setzen der Uhrzeit)
v$ = TIME$ (Lesen der Uhrzeit)

Bemerkungen:

Der von TIME$ ermittelte Wert entspricht natuerlich nur dann der tatsaechlichen Uhrzeit, wenn die Systemzeit beim Start des Computers korrekt eingegeben wurde.

Ueber die Funktion TIMER kann die Sekundenzahl seit dem Start des Computers bzw. die Tageszeit in Zentelsekunden, ueber MTIMER in Millisekunden ermittelt werden.

- MTIMER

Lesen und Setzen des Mikrosekunden-Zaehlers.

Syntax: **MTIMER** (Anweisung zum Setzen)
v = MTIMER (Funktion zum Lesen)

Bemerkungen:

Die **Anweisung** MTIMER setzt den Mikrosekunden-Zaehler des Computers auf den Wert 0 zurueck. Die **Funktion** MTIMER ermittelt die Anzahl von Mikrosekunden, die seit dem letzten MTIMER-Befehl vergangen sind.

Eine Kombination aus der Anweisung und der Funktion MTIMER ermoeglicht Zeitmessungen mit einer "typischen" Genauigkeit von rund zwei Mikrosekunden.

Der von der Funktion MTIMER zurueckgelieferte Wert hat die Einheit Millisekunde. Ein Wert wie 34.03 steht also fuer 34 Millisekunden und 30 Mikrosekunden.

Grenzen:
- MTIMER ist ausschliesslich fuer Messungen ueber sehr kurze Zeitraeume gedacht.

- Die Anweisungen SOUND und PLAY benutzen dasselbe Zaehlregister - eine gleichzeitige Verwendung zusammen mit MTIMER ist deshalb nicht moeglich.

Beispiel:

```
'Zuruecksetzen des Zaehlers
MTIMER

Pi# = 3*ATN(1)          'eine Berechnung

T% = MTIMER

PRINT "Die Berechnung der Zahl Pi hat ";_
      USING "##### Millisekunden gedauert.";T%
```

- **BEEP**

Der Lautsprecher erzeugt einen Ton.

Syntax: **BEEP [Anzahl]**

Bemerkungen:

Anzahl ist ein optionaler Wert und gibt an, wie oft der Ton ueber den Lautsprecher ausgegeben werden soll.

Beispiel:

```
BEEP 5          'Erzeugt 5 Toene
```

4.2.3. Typdeklarationsanweisungen DEFTyp

Definiert Variablentypen ueber das erste Zeichen ihres Namens.

Syntax: **DEFTyp [Buchstabenbereich [,Buchstabenbereich]...**

Bemerkungen:

Typ Eine Variable oder Funktion hat einen bestimmten Typ:
INT Integer
LNG Langinteger
SNG einfache Genauigkeit
DBL doppelte Genauigkeit
STR Zeichenkette

Buchstabenbereich

ist entweder ein einzelner Buchstabe (A...Z) oder ein Bereich, der durch zwei mit einem Bindestrich verbundene Buchstaben angegeben wird.

Unterschiede:
BASI macht entsprechende Typzuordnungen vom Fluss des Programms abhaengig, TBASIC dagegen von der Position der Anweisung im Quelltext.

Beispiel:

```
10 GOTO 30
20 DEFINT A-D
30 C = 75.3 : PRINT C
```

BASI arbeitet die Anweisung DEFINT in Zeile 20 nicht ab, die Variable C ist folglich vom Typ einfache Genauigkeit. Das Ergebnis des PRINT-Befehls ist 75.3. Der Compiler von TBASIC haelt sich dagegen nicht an den Fluss des Programmes, da dieser waehrend der Uebersetzung unbekannt ist. Da die Anweisung DEFINT im Quelltext vor der Zuordnung zur Variablen C steht, erhaelt die Variable das Format Integer. Dementsprechend gibt der PRINT-Befehl den Wert 75 aus.

Grenzen:
Die Anweisung **DEFTyp** gilt fuer alle Variablen- und Funktionsnamen, die mit einem der angegebenen Buchstaben beginnen. DEFINT B gibt also nicht nur der Variablen B den Typ Integer, sondern auch einer Funktion wie **Berechne A*B**. Die Angabe von Zeichenfolgen ist nicht moeglich - eine Anweisung wie DEFLNG AB bezieht sich leider nicht auf alle Bezeichner, die mit dem Buchstaben AB beginnen, sondern erzeugt einen Fehler beim Compilieren.

4.2.4. Wertzuweisungen

- LET

Ordnet den Wert eines Ausdrucks einer Variablen zu.

Syntax: **[LET] Variablenname = Ausdruck**

Bemerkungen:
keine

- SWAP

Vertauscht die Werte zweier Variablen.

Syntax: **SWAP Variable 1, Variable 2**

Bemerkungen:
keine

- INCR

Erhoeht eine Variable.

Syntax: **INCR Variablenname [,Wert]**

Bemerkungen:

Variablenname steht fuer den Namen einer numerischen Variablen, die erhoeht werden soll.

Wert steht fuer einen optionalen numerischen Ausdruck, der angibt, um welchen Wert diese Variable erhoeht werden soll. Ohne Angabe wird die Variable um 1 erhoeht.

Unterschied:
Dieser Befehl ist unter BASI nicht definiert.

- DECR

Verringert eine Variable.

Syntax: **DECR Variablenname [,Wert]**

Bemerkungen:

Variablenname steht fuer den Namen einer numerischen Variablen, die verringert werden soll.

Wert steht fuer einen optionalen numerischen Ausdruck, der angibt, um welchen Wert diese Variable verringert werden soll. Ohne Angabe wird die Variable um 1 verringert.

Beispiel:

```
I% = 10 : K% = 100
WHILE I% > 0
      PRINT I%, K%          'Anzeige beider Werte
      DECR I%               'I% um 1 verringern
      DECR K%, 5            'K% um 5 verringern
WEND
```

- RANDOMIZE

Setzt einen Startwert fuer einen Zufallszahlengenerator.

Syntax: **RANDOMIZE [n]**
RANDOMIZE TIMER/MTIMER

Bemerkungen:
keine

4.2.5. Dialogorientierte Ein-/Ausgabe

- INPUT

Eingabe eines oder mehrerer Werte ueber die Tastatur waehrend der Programmausfuehrung und Zuweisen an bestimmte Variablen.

Syntax: **INPUT [;][“Bedienerfuehrung”;|,]Variablenliste**

Bemerkungen:

Mit INPUT koennen maximal 255 Zeichen eingegeben werden.

Unterschiede:
In TBASIC dient nicht nur das Komma, sondern auch das Leerzeichen zur Trennung zwischen 2 Werten.

- LINE INPUT

Eingabe einer Zeile von der Tastatur in eine Zeichenketten-Variable.

Syntax: **LINE INPUT [;]["Bedienerfuehrung";]**
Zeichenkettenvariable

Bemerkungen:
keine

- PRINT

Ausgabe von Daten auf dem Bildschirm.

Syntax: **PRINT [Liste von Ausdruecken][;|,]**
? [Liste von Ausdruecken][;|,]

Bemerkungen:
keine

Unterschiede:
Bei BASI wird das Fragezeichen, das fuer PRINT eingegeben werden kann, sofort nach der Eingabe in PRINT gewandelt.
TBASIC laesst das Fragezeichen fuer PRINT im Quelltext unveraendert stehen.

- PRINT USING

Formatierte Ausgabe von Daten auf dem Bildschirm.

Syntax: **PRINT USING Formatzeichenkette; Liste von Ausdruecken [;]**

Bemerkungen:

Liste von Ausdruecken

enthaelt die formatiert auszugebenden Daten, die voneinander durch Leerzeichen, Komma oder Semikolon getrennt sind. Alle Trennzeichen haben die gleiche Wirkung.

Der Versuch, mehr als 24 Dezimalstellen auszugeben, endet mit einer Fehlermeldung.

Ignorieren von Format-Zeichen

Damit PRINT USING ein Format-Zeichen wie z.B. + oder ! in einer Formatzeichenkette als normales Zeichen ansieht, muss in diesem Fall ein Unterstrich dem Zeichen vorangestellt werden.

Beispiel:

```
"!" soll nicht als Zeichenketten-Formatzeichen gelten:
PRINT USING "Mark ####.## ist viel Geld_!"; 1000.00
```

- LPRINT und LPRINT USING

Ausgabe von Zeichen auf dem Drucker (LPT1:).

Syntax:
```
LPRINT [Liste von Ausdruecken][;]
LPRINT USING Formatzeichenkette;
             Liste von Ausdruecken[;]
```

Bemerkungen:
keine

- WRITE

Schreibt Daten auf den Bildschirm, getrennt durch Kommas.

Syntax:
```
WRITE [Liste von Ausdruecken]
```

Bemerkungen:

Wird eine Liste von Ausdruecken angegeben, wirkt WRITE wie PRINT, mit folgenden Ausnahmen:

- Nach der Ausgabe jedes Elements (ausser dem letzten) gibt WRITE ein Komma aus.
- Die Werte von Zeichenketten-Ausdruecken werden in Anfuehrungszeichen gesetzt.
- Positiven Zahlenwerten wird kein Leerzeichen vorangestellt.
- WRITE endet grundsaetzlich mit der Ausgabe eines Zeilenvorschubs.

- KEY

Setzt definierte Funktionstasten sowie Anzeige der Belegung auf dem Bildschirm.

Syntax:
```
KEY ON/OFF/LIST
KEY n, Zeichenkette
KEY n, CHR$(Tastenkennzeichen) + CHR$(Suchcode)
```

Bemerkungen:

- Mit den Anweisungen **KEY ON** und **KEY OFF** kann die Anzeige der Funktionstasten F1...F10 in der 25. Zeile des Bildschirms aktiviert bzw. geloescht werden.
  Die Definitionen selbst bleiben erhalten.
  KEY ON ist die Standardeinstellung.
  Solange die Belegung der Funktionstasten mit KEY ON aktiviert wird, werden nur die Zeilen 1...24 ueber den Bildschirm gerollt. Die Anzeige der Zeile 25 bleibt erhalten. Auf diese Zeile kann in dem Fall nicht mit LOCATE zugegriffen werden. Dies ist nur moeglich, wenn ueber KEY OFF die Anzeige der Funktionstasten geloescht wurde.
- Die Zuordnung einer **Zeichenkette** zu einer Taste **n** ist nicht nur fuer Funktionstasten moeglich.
  n ist ein Integer-Ausdruck im Bereich von 1..31 (Funktionstasten F1...F10, Kursortasten und selbstdefinierte Tastenfunktionen).
  Bei Druck auf eine derart definierte Taste reagiert das Programm so, als waere die entsprechende Tastenfolge manuell eingegeben worden.

- **KEY n, CHR$(Tastenkennzeichen) + CHR$(Suchcode)** ermoeglicht die Zuordnung der Nummer **n** zu einer Tastenkombination, z.B. ^A.
  n liegt im Bereich 15...29.

Tastenkennzeichen ist ein Integer-Ausdruck im Bereich 0..255 und legt ueber ein Bitmuster fest, welche Tasten zusaetzlich gedrueckt werden muessen, damit ON KEY reagiert.

Taste	Wert(Binaer)	Wert(Hex)
SHIFT rechts	0000 0001	01
SHIFT links	0000 0010	02
CTRL	0000 0100	04
ALT	0000 1000	08
NUM LOCK	0010 0000	20
CAPS LOCK	0100 0000	40

Suchcode ist ein numerischer Wert im Bereich von 1 bis 83 und bestimmt, welcher Taste mit KEY eine Nummer zugewiesen werden soll. Eine Tabelle der Suchcodes und der dazugehoerigen Tasten finden Sie in den Anlagen D und E. Die Funktionstasten sind bereits als "Tastennummern" 1...10, die Kursortasten als 11...14 definiert. Eine Neudefinition mit KEY hat deshalb keine Wirkung.

Der Druck auf eine Taste **n** kann vom Programm (unabhaengig von seinem sonstigen Verlauf) "abgefangen" werden: Mit der Anweisung **ON KEY (n) GOSUB** wird ein Unterprogramm festgelegt. Nach dem Aktivieren der Taste **n** mit **KEY (n) ON** prueft TBASIC vor der Ausfuehrung jedes Programmschritts, ob die Taste **n** gedrueckt wurde. Wenn ja, erfolgt ein Aufruf des durch **ON KEY (n) GOSUB** festgelegten Unterprogramms, bevor das "normale" Programm fortgesetzt wird. Mit **KEY (n) OFF** laesst sich diese Form wieder abschalten.

Beispiel:

Wenn ein Programm, unabhaengig von seinem normalen Ablauf, auf die Tastenkombination <SHIFT - ESC> in einer bestimmten Weise reagieren soll, dann muss wie folgt vorgegangen werden:

1. Ermitteln des Bitmusters fuer **Tastenkennzeichen**: Hier wird kein Unterschied zwischen der linken und der rechten SHIFT-Taste gemacht. Beide sollen zusammen mit <ESC> dieselbe Reaktion hervorrufen. Der Wert von **Tastenkennzeichen** muss also 3 (=1 + 2) betragen.

2. Ermitteln des Wertes fuer **Suchcode**. Aus Anlage E ist zu erkennen, dass der Taste <ESC> der Code 1 zugeordnet ist.

3. Zuordnen eines Wertes **n** zu der Tastenkombination <SHIFT-ESC>. Die niedrigste Nummer, die verteilt werden kann, ist 15 - die Nummern 1 bis 14 sind bereits vordefiniert und bezeichnen die Funktions- und Kursortasten. Im Beispiel wird unserer Tastenkombination also der Wert 15 zugeordnet:

   ```
   KEY 15, CHR$(&H03) + CHR$(1)
   ```

4. Jetzt muss nur noch festgelegt werden, welches Unterprogramm aufgerufen werden soll, wenn <SHIFT-ESC> gedrueckt wird:

   ```
   ON KEY(15) GOSUB ShiftEscape
   ```

5. Nun wird entschieden, dass die Routine auch wirklich aufgerufen werden soll und der Tastendruck nicht ignoriert wird:

   ```
   KEY(15) ON
   ```

   Ab diesem Zeitpunkt wird das Unterprogramm "ShiftEscape" automatisch aufgerufen, sobald <SHIFT-ESC> gedrueckt wird.

6. Ausschalten dieser Routine:

   ```
   KEY(15) OFF
   ```

- WIDTH

Legt die Breite einer Ausgabezeile des Bildschirms, eines Peripheriegeraetes oder einer Datei fest.

Syntax: **WIDTH [Geraetename,| Dateinummer,] Breite**

Bemerkungen:

Breite ist ein Integer-Ausdruck von 1 bis 255, der die gewuenschte Zeilenbreite angibt.

Geraetename ist ein (optionaler) Zeichenkettenausdruck, der bestimmt, fuer welches Peripheriegeraet die Zeilenbreite festgelegt werden soll. Zulaessige Angaben fuer **Geraetename** sind SCRN:, LPT1:, LPT2:, LPT3:, COM1: und COM2:.
Wenn weder ein Geraetename noch eine Dateinummer angegeben ist, dann bezieht sich WIDTH auf den Bildschirm (SCRN:).

Dateinummer ist ein (optionaler) Integer-Ausdruck und stellt die Nummer einer Datei dar, deren "Zeilenbreite" gesetzt werden soll. Die Datei muss im Modus OUTPUT eroeffnet worden sein.

Im allgemeinen legt WIDTH fest, wieviele Zeichen nacheinander ausgegeben werden koennen, bevor automatisch ein Zeilenvorschub erfolgt. Die tatsaechliche Wirkung der Anweisung ist aber davon abhaengig, auf welches Ausgabegeraet sie sich bezieht.

Der Befehl LPRINT macht hier eine Ausnahme: Jede Ausgabe zu einem angeschlossenen Drucker beinhaltet ein implizites OPEN (und danach ein CLOSE) - WIDTH zeigt also bereits beim naechsten LPRINT-Befehl eine Wirkung.

WIDTH #Dateinummer, Breite aendert die logische Zeilenbreite der entsprechenden Datei mit sofortiger Wirkung. Einen Sinn hat dieser Befehl jedoch nur, wenn die Datei einem der Geraete LPT1:, LPT2:, LPT3:, COM1: oder COM2: zugeordnet ist. Bei Diskettendateien tut sich nichts.

4.2.6. Arbeit mit dem Konstantenspeicher

- READ

Liest Elemente von DATA-Anweisungen und weist diese Werte Variablen zu.

Syntax: **READ Variable[,Variable...]**

Bemerkungen:

TBASIC verfuegt ueber einen "DATA-Zeiger", der nach dem Start des Programms auf die erste mit DATA definierte Konstante des Programms zeigt. Jeder READ-Befehl liest eine oder mehrere DATA-Elemente und erhoeht den Zeiger entsprechend. Der Versuch, den DATA-Zeiger ueber das letzte DATA-Element des Programms hinauszubewegen (d.h. mehr READ-Befehle als DATA-Elemente vorhanden sind), erzeugt den Laufzeitfehler 4.

Mit **RESTORE** kann der DATA-Zeiger wieder auf das erste (oder ein bestimmtes) Element zurueckgesetzt werden. Ein Neustart des Programms mit RUN oder der Befehl CLEAR setzen den DATA-Zeiger immer auf das erste DATA-Element zurueck.

- DATA

Definieren von Konstanten fuer READ-Anweisungen.

Syntax: **DATA Konstante [,Konstante...]**

Bemerkungen:

Eine der Systemvariablen von TBASIC ist der sogenannte "DATA-Zeiger". Er zeigt direkt nach dem Start des Programms auf das erste DATA-Element und wird durch jede READ-Anweisung auf das jeweils naechste DATA-Element weiterbewegt. Durch die Befehle **RUN, CLEAR** und **RESTORE** wird der DATA-Zeiger wieder auf das erste DATA-Element zurueckgesetzt.

RUN und **CLEAR** loeschen zwar auch saemtliche Variablen, die durch das Lesen von DATA-Elementen gesetzt worden sind, die DATA-Elemente selbst bleiben aber von diesen Befehlen unberuehrt und koennen danach mit READ erneut gelesen werden.

DATA-Anweisungen sind fuer den Ablauf des Programms "unsichtbar", koennen also an beliebiger Stelle zwischen anderen Befehlen stehen. Ein "Durchlaufen" der DATA-Anweisungen ist nicht notwendig - direkt nach dem Start des Programms stehen saemtliche Elemente unabhaengig von ihrer Position im Quelltext zur Verfuegung.

- **RESTORE**

Setzt den DATA-Zeiger des Programms auf das erste oder auf ein bestimmtes Element.

Syntax: **RESTORE [Label]**

Bemerkungen:

RESTORE ohne weitere Angaben setzt den DATA-Zeiger auf das erste DATA-Element des Programms und hat somit dieselbe Wirkung wie CLEAR oder der Neustart des Programms mit RUN.

Durch Angabe des (optionalen) Parameters **Label** kann der DATA-Zeiger auf die DATA-Anweisung gesetzt werden, die auf **Label** folgt. Der naechste READ-Befehl liest dann ab dieser DATA-Anweisung weiter.

4.2.7. Steueranweisungen

- **STOP**

Bricht die Programmausfuehrung ab.

Syntax: **STOP**

Bemerkungen:

STOP beendet ein Programm in derselben Weise wie END. Saemtliche eventuell offenen Dateien werden geschlossen. Danach folgt ein Ruecksprung auf die Kommandoebene von DCP bzw. zur Programmierumgebung von TBASIC, je nachdem, von wo aus das Programm gestartet wurde.

Im Gegensatz zu BASI ist ein Fortsetzen des Programms nicht moeglich. Der Befehl **CONT** ist in TBASIC nicht implementiert. Eine Meldung ueber den Punkt des Abbruchs (Zeilennummer, Programmzaehler o.ae.) wird ebenfalls nicht angezeigt.

STOP sollte aus diesen Gruenden lediglich waehrend der Testphase eines Programms und die Anweisung END in einem gueltigen Programm verwendet werden.

- DELAY

Haelt die Programmausfuehrung fuer einen definierten Zeitraum an.

Syntax: **DELAY Sekunden**

Bemerkungen:

Sekunden ist ein numerischer Ausdruck und gibt die Zeit in Sekunden an, fuer die das Programm vor der Ausfuehrng des naechsten Befehls angehalten werden soll.

Dieser Parameter wird als Zahl einfacher Genauigkeit behandelt und darf folglich auch Nachkommastellen enthalten. Die maximale Aufloesung von DELAY liegt im Bereich von 0.054 Sekunden.

Der Befehl DELAY sollte grundsaetzlich anstelle von Leerschleifen zum Verzoegern verwendet werden, weil er unabhaengig von der Wortbreite und der Taktfrequenz des verwendeten Prozessors arbeitet.

DELAY laesst sich auch nicht mit <CTRL-BREAK> unterbrechen.

Beispiel:

```
PRINT "Ein Tastendruck beendet das Programm..."
WHILE NOT INSTAT        'solange kein Tastendruck
      LOCATE 10,30
      PRINT TIME$       'Ausgabe der Uhrzeit
      DELAY 5           'und 5 Sekunden warten
WEND
END
```

- END

Beendet die Strukturdefinition oder die Programmausfuehrung.

Syntax: **END [DEF|IF|SELECT|SUB]**

Bemerkungen:

Die Anweisung END ohne Parameter beendet die Ausfuehrung des Programms.
Der TBASIC-Compiler orientiert sich (im Gegensatz zu einigen anderen Sprachen wie z.B. PASCAL) ausschliesslich am physikalischen Ende des Quelltextes - eine END-Anweisung beendet also den Uebersetzungsvorgang nicht. Darueber hinaus darf ein Programm eine beliebige Anzahl von END-Anweisungen enthalten.

Ein **END** am Ende des Hauptprogramms ist vor allem dann erforderlich, wenn sich hinter dem Hauptprogramm Unterprogramme befinden, die sonst nachfolgend abgearbeitet werden.
Mit **END** werden saemtliche offenen Dateien geschlossen. Danach erfolgt ein Ruecksprung zu DCP bzw. der Programmierumgebung von TBASIC.

Zusammen mit **DEF, IF, SUB** oder **SELECT** kennzeichnet **END** das Ende einer dieser vier Strukturen. Details dazu sind den entsprechenden Abschnitten dieses Kapitels zu entnehmen.

- GOTO

Es wird ein unbedingter Sprung ausgefuehrt.

Syntax: **GOTO Bezeichner**

Bemerkungen:

Bezeichner ist eine Konstante (Label oder Zeilennummer) und bezeichnet den Punkt, an dem die Programmausfuehrung fortgesetzt werden soll.

Im Gegensatz zu GOSUB findet bei GOTO kein Speichern des momentanen Programmzaehlerstandes statt. **"Unbedingt"** bedeutet hier, dass GOTO immer einen Sprung unabhaengig von eventuell weiteren Bedingungen ausfuehrt. Natuerlich kann man durch ein vorangestelltes IF/THEN dafuer sorgen, dass der GOTO-Befehl selbst nur in Abhaengigkeit von einer bestimmten Bedingung ausgefuehrt wird.
Der Befehl GOTO wird von den Verfechtern anderer Programmiersprachen als der Punkt angesehen, der BASIC zu einer "unmoeglichen" Sprache macht. Vernueftig eingesetzt erspart einem ein GOTO dagegen vierte und fuenfte IF-Ebenen und/oder Block-Strukturen, die im Listing ueber mehrere Seiten gehen.

Beispiel:

Die beiden folgenden Programmteile erfuellen exakt dieselbe Aufgabe - der eine ist mit GOTO, der andere ohne GOTO programmiert.

```
'Programm 1: mit GOTO
'Die hier nicht weiter ausgefuehrten Prozeduren
'DatenLesen und DatenSchreiben setzen die
'Variable Fehler% im Fehlerfall auf <> 0.
'Dieses Programm waere unter Verwendung von
'GOSUB und RETURN <Label> noch wesentlich
'kuerzer ausgefallen!

Fehler% = 0
CALL DatenLesen: IF Fehler% THEN GOTO Abbruch
CALL Eingabe: IF Fehler% THEN GOTO Abbruch
CALL DatenSchreiben
     IF Fehler% THEN GOTO Abbruch

PRINT "Programm fehlerfrei beendet."
END
 .
 .
 .
Abbruch:
   PRINT "Fehler! " : PRINT "Programm abgebrochen"
END
```

```
'Programm 2: ohne GOTO    Benutzt dieselben
'Prozeduren wie Programm 1
'Programm ist komplizierter
Fehler% = 0
CALL DatenLesen
IF NOT Fehler% THEN
   CALL Eingabe
   IF NOT Fehler% THEN
      CALL DatenSchreiben
      IF NOT Fehler% THEN
         PRINT "Programm fehlerfrei beendet."
      END IF
   ELSE
      PRINT "Fehler! Programm abgebrochen."
   END IF
ELSE
   PRINT "Fehler! Programm abgebrochen."
END IF
```

- GOSUB...RETURN

Aufruf eines BASIC-Unterprogramms.

Syntax: **GOSUB Bezeichner**

Bemerkungen:

Bezeichner ist eine Konstante (entweder ein Label oder eine Zeilennummer) und bestimmt den Startpunkt des aufzurufenden Unterprogramms.

TBASIC speichert den momentanen Stand des Programmzaehlers (auf dem Stack) und fuehrt dann einen **"unbedingten"** Sprung zu dem Befehl aus, der unter **Bezeichner** folgt.

Ein RETURN innerhalb des aufgerufenen Unterprogramms bewirkt einen Ruecksprung. Das Programm wird an der Stelle fortgesetzt, die sich unmittelbar hinter dem GOSUB-Befehl befindet.

An dieser Stelle muss darauf hingewiesen werden, dass die Erweiterungen von TBASIC das alte Unterprogramm-Konzept von BASI mehr oder weniger ueberfluessig machen. Unterprogramme koennen im Gegensatz zu Prozeduren und Funktionen weder lokale Variablen noch Parameter enthalten. Rekursive Definitionen sind ebenfalls nicht moeglich. Ein Unterprogramm ist darueber hinaus in keiner Weise vom restlichen Programmcode isoliert (siehe END).

Im Gegensatz zu Prozedur- und Funktionsdefinitionen koennen Unterprogramme allerdings so tief geschachtelt werden, wie es die Groesse des Stacks zulaesst. Ein Unterprogramm kann nicht nur andere Unterprogramme aufrufen, sondern auch selbst weitere Unterprogramm-Ebenen enthalten.

Ausserdem werden Unterprogrammstrukturen fuer alle Hintergrundprozesse von TBASIC (PLAY, dem "Abfangen" empfangener Zeichen ueber die seriellen Ports usw.) verwendet.

Vorsicht mit ueberzaehligen RETURNs: TBASIC holt bei einem RETURN-Befehl die jeweils oberste Rueckspungadresse vom Stack herunter und fuehrt einen entsprechenden Sprung ohne weitere Pruefungen aus - es sei denn, der Schalter **Stack test** im Menue **Options** ist vor dem Compilieren gesetzt worden. In diesem Fall wird folgende Fehlermeldung angezeigt:

Fehler 3 RETURN ohne GOSUB, pgn-ctr...

Wenn dieser Schalter auf OFF steht, dann stuerzt das System bei undefinierten Rueckspruengen ab. Waehrend der Entwicklungsphase eines Programms muss **Stack test** grundsaetzlich auf ON gesetzt werden!

- ON GOTO / ON GOSUB

Ruft abhaengig vom Wert eines numerischen Ausdrucks eines von mehreren Programmteilen bzw. eines von mehreren Unterprogrammen auf.

Syntax: **ON n GOTO Label [,Label...]**
ON n GOSUB Label [,Label...]

Bemerkungen:

n ist ein numerischer Ausdruck im Bereich 0...255.

Label bezeichnet den Startpunkt der jeweiligen Programmteile bzw. Unterprogramme. Theoretisch koennen durch Verlaengerung der Zeile mir einem Unterstrich bis zu 255 Bezeichner angegeben werden.

ON/GOTO ruft abhaengig vom Wert **n** einen Programmteil und **ON/GOSUB** ein Unterprogramm der "Label-Liste" auf. Wenn **n** den Wert 1 hat, wird der zuerst genannte Programmteil bzw. das zuerst genannte Unterprogramm angesprungen bzw. aufgerufen, fuer n = 2 das 2. usw.

Wenn **n** groesser als die Anzahl der angegebenen Labels ist, wird die Ausfuehrung des Programms mit dem naechsten Befehl nach ON/GOTO bzw. ON/GOSUB fortgesetzt.
Dasselbe gilt fuer n = 0.

Jedes der aufgefuehrten Unterprogramme sollte mit RETURN enden. Die Programmausfuehrung wird dann mit dem naechsten Befehl nach ON/GOSUB fortgesetzt.

Der Wert angegebener Zeilennummern bzw. die Position der einzelnen Programmteile im Quelltext ist unerheblich - entscheidend ist nur die Reihenfolge der Labels in der Liste.

Mit SELECT und einer IF-Blockstruktur koennen Mehrweg-Verzweigungen in TBASIC wesentlich flexibler realisiert werden als mit dem traditionellen ON/GOTO bzw. ON/GOSUB.

Beispiel:

```
FOR i% = 0 TO 4
ON i% GOTO Routine1, Routine2, Routine3
PRINT "Kein Sprung. i% = "; i%

Zurueck:
   NEXT i%
END             'Programmende
```

```
Routine1:
   PRINT "Hier Routine1."
   GOTO Zurueck

Routine2:
   PRINT "Hier Routine2."
   GOTO Zurueck

Routine3:
   PRINT "Hier Routine3."
   GOTO Zurueck
```

- RETURN

Beendet ein Unterprogramm und fuehrt einen Ruecksprung aus.

Syntax: **RETURN [Label]**

Bemerkungen:

RETURN ohne weitere Parameter fuehrt zu einem Ruecksprung ins Hauptprogramm. TBASIC setzt die Abarbeitung mit dem Programmschritt fort, der unmittelbar auf das GOSUB folgt.

Wenn der (optionale) Parameter **Label** angegeben ist, wird die Programmausfuehrung nach dem Ruecksprung mit dem auf Label folgenden Befehlen fortgesetzt. Die Fehlermeldung "RETURN ohne GOSUB" wird nur angezeigt, wenn vor der Compilierung der Schalter **Stack test** im Menue **Options** auf ON gesetzt wurde. Dieser Schalter sollte waehrend der Entwicklungsphase eines Programms grundsaetzlich auf ON gesetzt sein.

Beispiel:

Das folgende Programmfragment wird in zwei Arten programmiert:

```
'Version 1 mit RETURN <Label>

'**** Hauptprogramm ****
 .
 .
 .
GOSUB DatenEinlesen
GOSUB Auswahl
GOSUB DatenSchreiben
GOTO Ende

'Dieser Teil wird mit RETURN <Label> erreicht
Fehler:
      PRINT "Das Programm wurde abgebrochen!

Ende:
END              'Programmende
```

```
DatenEinlesen:
   OPEN "ADRESS.DAT" FOR INPUT AS #1
   INPUT #1, Anzahl%
   IF Anzahl% = 0 THEN RETURN Fehler
   .
   .
   .
RETURN

Auswahl:
   .
   .
   .
RETURN

DatenSchreiben:
   IF Anzahl% = 0 THEN RETURN Fehler
   OPEN "ADRESS.DAT" FOR OUTPUT AS #1
   .
   .
   .
RETURN

'Version 2 - die herkoemmliche Loesung
'ohne GOTO und ohne RETURN <Label>

'**** Hauptprogamm ****

RICHTIG = -1   'TRUE
   .
   .
   .
GOSUB DatenEinlesen
IF RICHTIG THEN
   GOSUB Auswahl
   IF RICHTIG THEN
      GOSUB DatenSchreiben
   END IF
ELSE
   PRINT "Programm abgebrochen!"
END IF
    IF RICHTIG THEN
          PRINT "Programm beendet."
    END IF
END        'Programmende
```

```
DatenEinlesen:
      OPEN "ADRESS.DAT" FOR INPUT AS #1
      INPUT #1, Anzahl%
      IF Anzahl% = 0 THEN
         RICHTIG = 0        'FALSE
      ELSE ...
      .
      .
      END IF
RETURN
Auswahl:
    .
    .
    .
RETURN

DatenSchreiben:
    IF Anzahl% = 0 THEN
       RICHTIG = 0          'FALSE
    ELSE:
       OPEN "ADRESS.DAT" FOR OUTPUT AS #1
    .
    .
    END IF
RETURN
```

FOR ... NEXT

Fuehrt einen Zyklus von Anweisungen aus, wobei die Anzahl der Durchlaeufe ueber eine Laufvariable festgelegt wird, die bei jedem Durchlauf automatisch erhoeht bzw. verringert wird.

Syntax:

```
FOR  Variablenname = x TO y [ STEP z ]
  .
  .          Anweisungen
  .
NEXT [ Variablenname [, Variablenname ... ]
```

Bemerkungen:

Variablenname steht fuer den Namen einer numerischen Variablen, die als Laufvariable der Schleife bezeichnet wird.

x,y und z sind numerische Ausdruecke, mit denen die Start- und Abbruchbedingungen der Schleife festgelegt werden.

TBASIC fuehrt FOR / NEXT - Schleifen dann am schnellsten aus, wenn die Laufvariable als Integervariable und x,y,z als Konstanten angegeben werden.

FOR / NEXT - Schleifen belegen keinen Platz auf dem Stack und koennen deshalb bis in sehr grosse Tiefen verschachtelt werden. Voraussetzung fuer jede Verschachtelung ist, dass sich innere und aeussere Schleifen nicht ueberkreuzen.

Hinweis:

Auch wenn der Compiler Leerzeichen grundsaetzlich ignoriert - es kommt der Lesbarkeit eines Programms ausserordentlich zugute, die Schleifenanweisungen im Quelltext um zwei oder drei Leerzeichen einzuruecken.

Mit dem Befehl EXIT FOR kann eine Schleife verlassen werden, bevor ihre durch den Endwert gesetzte Abbruchbedingung erfuellt ist. Der Befehl GOTO sollte an dieser Stelle nicht eingesetzt werden, auch wenn er hier zulaessig ist. Nicht erlaubt ist da-gegen das Hineinspringen in eine Schleife: Ein fuer das Programm unerwartetes NEXT fuehrt zum Laufzeitfehler 1 (NEXT ohne FOR).

Hinweis:

Zu jedem FOR gehoert nur ein NEXT. Das folgende Beispiel endet mit einem Fehler bei der Compilierung:

```
FOR x% = 1 TO 5
    IF x% = < THEN NEXT: GOTO 99
    PRINT "x ist ungleich 4"
NEXT x%

99 PRINT "Programmende"
```

Hier erkennt der Compiler die erste NEXT-Anweisung (nach THEN) als Ende der Schleife und meldet beim zweiten NEXT folgerichtig, dass ihm ein dazugehoeriges FOR fehlt. Die korrigierte Version dieses Beispiels lautet:

```
FOR x% = 1 TO 5
    IF x% = 4 THEN GOTO 99
    PRINT "x ist ungleich 4"
99 NEXT

PRINT "Programmende"
```

Grenzen:
Eine FOR/NEXT-Schleife mit der Schrittweite 0 stellt keinen illegalen Funktionsaufruf dar und wird vom Compiler ebenfalls nicht beanstandet. Falls die Schleifenanweisungen nicht den Befehl EXIT FOR enthalten, wird kein Ende der Schleife erreicht.

Eine Laufvariable unterscheidet sich in keiner Weise von einer "normalen" Variablen. Theoretisch ist es deshalb moeglich, ihr **innerhalb** einer Schleife einen anderen Wert zuzuweisen.

Unterschiede:
Der BASIC-Interpreter unterscheidet zwischen NEXT-Anweisungen mit und solchen ohne angegebenen Variablennamen. Im ersten Fall wird die Variablentabelle nach dem angegebenen Variablennamen durchsucht (was je nach Groesse der Tabelle eine gewisse Zeit dauert), im zweiten Fall einfach die zuletzt auf den Stack gelegte Laufvariable entsprechend erhoeht/erniedrigt. Da die Lokalisierung von Laufvariablen bei TBASIC waehrend des Compilierens ausgefuehrt wird, ergibt sich hier kein Unterschied in der Ausfuehrungsgeschwindigkeit der Schleife.

- IF

Pruefen einer Bedingung und Aendern des Programmablaufs in Abhaengigkeit von der Bedingung.

Syntax: **IF Integer-Ausdruck[,] THEN Anweisung(en)**
[ELSE Anweisung(en)]

Bemerkungen:

Vor dem reservierten Wort ELSE darf kein Doppelpunkt stehen. Eine IF-Anweisung wie die folgende endet mit einer Fehlermeldung des Compilers:

```
IF A < B  THEN  C = D : ELSE  E = F
```

Die gesamte IF-Anweisung muss (inklusive THEN, ELSE und allen dazugehoerigen Befehlen) auf einer logischen Zeile stehen. Aus diesem Grund ergibt auch die folgende Programmierung einen Uebersetzungsfehler:

```
IF x>10 THEN PRINT "x ist groesser als 10"
ELSE PRINT "x ist kleiner als 10"
```

Der Compiler interpretiert den Zeilenvorschub am Ende der ersten Zeile als Ende der IF-Anweisung und versucht dann, die zweite Zeile als eigenstaendige Anweisung abzuarbeiten. Falls die 80 Spalten einer Zeile nicht ausreichen, kann auch hier wieder der Unterstrich (_) als Verlaengerung der Zeile benutzt werden:

```
IF x>10 THEN PRINT "x ist groesser als 10":_
   PRINT "Aber nicht mehr lange!" : x=0_
ELSE PRINT "x ist kleiner als 10"
```

Die bessere Alternative zu mehrfach verlaengerten Zeilen ist allerdings eine IF - Blockstruktur, welche bei BASI nicht vorhanden ist.

IF fuehrt zur Ermittlung des Wahrheitswertes eine Konvertierung in das Integerformat durch. Das Ergebnis eines Vergleichs hat entweder den Wert -1 oder den Wert 0, liegt also immer innerhalb des Geltungsbereiches fuer Integer-Variablen. Anders dagegen bei der direkten Auswertung von Konstanten:
Hier muss darauf geachtet werden, dass die angegebenen Werte im Bereich von -32768 bis +32767 liegen. Das folgende Beispiel erzeugt den Laufzeitfehler 6 (Ueberlauf) beim Auswerten der Bedingung:

```
x = 50000
IF x THEN PRINT "Ueberlauf"
```

- IF-Blockstruktur

Pruefen einer Serie von Vergleichen als Blockstruktur und Aendern des Programmablaufs in Abhaengigkeit von der Bedingung.

Syntax:

```
IF Integer-Ausdruck[,] THEN
      .
      .
      .
   Anweisung(en)
      .
      .
[ELSEIF Integer-Ausdruck[,] THEN
      .
      .
      .
   Anweisung(en)]
[ELSE
      .
      .
      .
   Anweisung(en)]
      .
      .
END IF
```

Bemerkungen:

Diese Blockdefinition stellt eine Erweiterung des Sprachumfangs von BASIC dar - sie erlaubt ganze Serien von Pruefungen, die sich ueber eine beliebige Zahl logischer Zeilen hinweg erstrecken koennen.

Bei der Ausfuehrung einer blockstrukturierten IF-Anweisung wird zuerst der Wahrheitswert des Ausdrucks hinter dem IF ermittelt. Ergibt sich der Wert FALSE, dann werden die Bedingungen der folgenden ELSEIF-Anweisungen in der Reihenfolge ihrer Programmierung ausgewertet (ein IF-Block kann eine beliebige Anzahl von ELSEIFs enthalten - oder auch gar keins).

Wenn sich einer der Werte als TRUE ergibt, werden die darauf folgenden Anweisungen ausgefuehrt. Danach folgt ein Sprung zu der Anweisung, die als naechstes hinter END IF steht. Anweisungen, die nach dem (optionalen) ELSE stehen, werden nur dann ausgefuehrt, wenn alle vorherigen Tests das Ergebnis FALSE hatten.

Achtung:
Der Compiler unterscheidet zwischen einfachen und blockstrukturierten IF-Anweisungen ueber den Inhalt der Zeile, in der sich das erste IF befindet. Ent-haelt diese Zeile nach dem THEN einen oder mehrere Befehle, wird IF als "einfach" bewertet. In einer IF-Blockstruktur muss die erste logische Zeile also mit dem THEN enden bzw. darf nach dem THEN nur noch ein Kommentar stehen. Dasselbe gilt fuer die Zeilen mit ELSEIF und ELSE:

```
IF x% = 4 THEN              'nichts nach THEN erlaubt
   PRINT: PRINT             'beliebige Befehlsfolgen
ELSEIF x%>4 THEN            'nichts nach THEN erlaubt
   PRINT: PRINT             'beliebige Befehlsfolgen
ELSE                        'nichts nach ELSE erlaubt
   PRINT: PRINT             'beliebige Befehlsfolgen
END IF
```

IF-Blocks koennen in grosse Tiefen verschachtelt werden. Jeder Befehl nach jedem THEN innerhalb eines IF-Blocks kann der Beginn eines weiteren IF-Blocks sein.
Ein IF-Block muss mit der Anweisung END IF abgeschlossen werden.

- WHILE / WEND

Fuehrt einen Zyklus von Anweisungen aus, wobei die Anzahl der Durchlaeufe variabel ist, abhaengig von einer angegebenen Bedingung.

Syntax:

```
WHILE Integer-Ausdruck
      .
      . Anweisung(en)
      .
WEND
```

Bemerkungen:

Die Abbruchbedingung wird am Schleifenanfang geprueft.

- DO / LOOP

Zyklische Abarbeitung einer Schleife mit Test auf TRUE / FALSE am Beginn und / oder am Ende der Schleife.

Syntax:

```
DO [WHILE/UNTIL Ausdruck]
   .
   . Anweisungen [EXIT LOOP]
   .
LOOP [WEND] [WHILE/UNTIL Ausdruck]
```

Bemerkungen:

Ausdruck steht fuer einen numerischen Ausdruck, der nur danach bewertet wird, ob er TRUE (ungleich 0) oder FALSE (0) ergibt.

Zwischen den reservierten Woertern DO und LOOP eingeschlossene Anweisungen werden bei der Ausfuehrung so lange wiederholt, bis eine Abbruchbedingung der Schleife erfuellt ist. Diese Bedingung kann sowohl am Anfang als auch am Ende der Schleife stehen. Im ersten Fall werden die Befehle innerhalb der Schleife komplett uebersprungen, wenn sie erfuellt ist. Im zweiten Fall wird die Schleife grundsaetzlich einmal durchlaufen und danach geprueft, ob sie wiederholt werden soll.

Ein Programmieren mit DO / LOOP ermoeglicht ausserdem eine Pruefung sowohl zu Beginn als auch am Ende der Schleife. Wenn keine Abbruchbedingung vorhanden ist (kein EXIT-Befehl), dann laeuft die Schleife endlos.

Zu jedem DO gehoert ein LOOP - auf Schleifen ohne erkennbares Ende reagiert der Compiler mit einer Fehlermeldung.

Der Befehl EXIT LOOP beendet eine Schleife mit sofortiger Wirkung und kann innerhalb der Schleife beliebig oft programmiert werden.

Mit **WHILE** und **UNTIL** wird die Pruefung einer Abbruchbedingung formuliert. WHILE wiederholt die Schleife, wenn der danach folgende Ausdruck den Wert TRUE hat (Wiederholen, solange die Bedingung erfuellt ist). UNTIL hat die entgegengesetzte Wirkung (Wiederholen, bis die Bedingung erfuellt ist).

Mit DO / LOOP aufgebaute Schleifen belegen keinen Platz auf dem Stack und koennen bis in beliebige Tiefen verschachtelt werden.

Beispiel:

Die folgende Schleife wird nur dann ausgefuehrt, wenn die Variable A den Wert 10 hat. Sie wird solange wiederholt, bis A einen Wert <> 10 bekommt:

```
DO WHILE A = 10
     .
     .  Anweisungen
     .
LOOP
```

Das naechste Beispiel benutzt UNTIL und verhaelt sich genau umgekehrt. Die Schleife wird nur dann ausgefuehrt, wenn A einen Wert <> 10 hat. Wiederholt wird sie solange, bis diese Variable den Wert 10 erhaelt:

```
DO UNTIL A = 10
   .
   .  Anweisungen
   .
LOOP
```

Eine Pruefung am Schleifenende bewirkt, dass die von DO und LOOP eingeschlossenen Befehle grundsaetzlich einmal ausgefuehrt werden. Derartige Strukturen eignen sich recht gut fuer Menues:

```
DO
     CLS
     PRINT " 1 - Programm 1 aufrufen "
     PRINT " 2 - Programm 2 aufrufen "
     PRINT " .... "
     PRINT " 9 - Programmende "
     PRINT
     INPUT " Bitte waehlen Sie: "; Wahl$
     ON VAL(Wahl$) GOSUB Prog1, Prog2 'Prog3 usw.
     DELAY 2
UNTIL Wahl$ = "9"
END

Prog1:
     PRINT "Hier Programm 1"
RETURN

Prog2:
     PRINT "Hier Programm 2"
RETURN
```

- SELECT

Bilden einer Blockstruktur fuer Pruefungen mit mehreren Moeglichkeiten.

Syntax:

```
SELECT CASE Ausdruck
   CASE Pruefungen
     .
     .  Anweisung(en)
     .
   [CASE Pruefungen
     .
     .  Anweisung(en)
     .                ] ...
   [CASE ELSE
     .
     .  Anweisung(en)
     .                ]
END SELECT
```

Bemerkungen:

Ausdruck ist ein numerischer oder ein Zeichenketten-Ausdruck. Sein Wert wird durch die folgenden **Pruefungen** getestet.

Wenn eine Pruefung den Wert TRUE ergibt (d.h. die dort definierten Bedingungen fuer den Wert von **Ausdruck** zutreffen), dann werden die darauffolgenden Anweisungen ausgefuehrt, anschliessend folgt ein Sprung zum naechsten Programmschritt nach **END SELECT**. Die Folge der **Pruefungen** ist durch ihre Reihenfolge im Quelltext festgelegt. Der durch das (optionale) CASE ELSE definierte Teil wird nur dann ausgefuehrt, wenn keine der vorherigen Pruefungen das Ergebnis TRUE hatte.

Innerhalb eines SELECT-Blocks kann eine beliebige Anzahl von **Pruefungen** definiert werden. Ein Verschachteln von SELECT-Bloecken (also weitere SELECT-Anweisungen innerhalb eines CASE-Zweiges) ist erlaubt.

Der (optionale) Befehl EXIT kann beliebig oft innerhalb der CASE-Zweige programmiert werden und kommt einem Sprung zum naechsten Befehl hinter END SELECT gleich (der theoretisch auch mit GOTO moeglich waere).

Fuer **Pruefungen** innerhalb von SELECT stehen eine Reihe von Moeglichkeiten zur Verfuegung, die beliebig miteinander kombiniert werden koennen.

Zwei Einschraenkungen sind zu beachten:

- Da jeder Test den Wert von **Ausdruck** prueft, muessen alle Tests denselben Typ (entweder numerisch oder Zeichenkette) haben. Eine SELECT - Anweisung, die beispielsweise zuerst prueft, ob **Ausdruck** eine Laenge groesser 6 hat und dann, ob das erste Zeichen von **Ausdruck** ein "A" ist, kann somit nicht programmiert werden.

- Saemtliche zur Pruefung verwendeten Werte muessen als Konstanten definiert sein. Eine Anweisung wie CASE MID$(4,1)="A" ist nicht erlaubt. Fuer solche Faelle sollte das blockstrukturierte IF verwendet werden.

Die folgende Liste gibt die moeglichen Tests innerhalb von **Pruefungen** wieder, das Zeichen K steht dabei jeweils fuer eine Konstante, die je nach Typ von **Ausdruck** numerisch sein oder den Typ Zeichenkette haben kann:

```
CASE < K                    'relationaler Vergleich
CASE K                      'Gleichheit
CASE K1 TO K2               'Bereich
CASE K1, K2                 'zwei Pruefungen auf Gleichheit

CASE K1 TO K2, K3           'Pruefen auf Bereich,
                            'Gleichheit
```

Mehrere Tests innerhalb einer Pruefung werden von SELECT mit einem OR verbunden. Eine Anweisung wie CASE K1 TO K2, K3 liesse sich also ueber eine IF - Bedingung folgendermassen formulieren:

```
IF(x>=K1 AND x<=K2) OR x = K3 THEN ...
```

Es muss beachtet werden, dass entweder mindestens eine der **Pruefungen** zutrifft oder ein CASE ELSE - Zweig programmiert wurde. Per Definition ist festgelegt, dass SELECT mit einem Laufzeitfehler endet, wenn keiner der Zweige ausgefuehrt werden kann (d.h. alle **Pruefungen** fehlschlagen und kein CASE ELSE-Zweig vorhanden ist).

Beispiel:

```
INPUT "Geben Sie eine Zahl ein: ",X

SELECT CASE X

     CASE < 20              '"IF X < 20 THEN .."
        PRINT "Kleine Zahl"

     CASE < 200             '"ELSEIF X < 200 THEN ... "
        PRINT "Mittlere Zahl"

     CASE 1111, 2222, 3333, 4444
        PRINT "Gleiche Zahlen"
        IF X = 4444 THEN                    'ein IF-Block
          PRINT "Zahl = 4444"               'in SELECT ist
        ELSE                                'moeglich
          PRINT "ungleich 4444"
        END IF

     CASE ELSE                              'ansonsten ...
        PRINT "Unbekannte Zahl"
END SELECT
```

- EXIT

Verlaesst eine Befehlsstruktur vor dem eigentlichen Ende, d.h., es werden eine oder mehrere Anweisungen uebersprungen.

Syntax: **EXIT SELECT | DEF | FOR | IF | LOOP | SUB**

Bemerkungen:

Mit EXIT kann ein Befehls-Block bzw. eine Prozedur- oder Funktionsdefinition vorzeitig verlassen werden. Die Art des zu verlassenden Blocks muss durch einen Zusatz zu EXIT angegeben werden:

Zusatz	**dadurch verlassene Struktur**
SELECT	SELECT-Befehlsblock
DEF	Funktionsdefinition
FOR	FOR / NEXT-Schleife
IF	IF-Befehlsblock
LOOP	DO / LOOP-Schleife
SUB	Prozedurdefinition

Ein Teil dieser Strukturen (speziell Funktions- und Prozedurdefinitionen) darf nur ueber EXIT verlassen werden. In allen anderen Faellen ist EXIT einem Sprung mit GOTO grundsaetzlich vorzuziehen.

Hinweis:

Beim Verlassen von Funktionsdefinitionen mit EXIT ist darauf zu achten, dass der Funktion vorher ein Ergebnis zuzuweisen ist - ansonsten ist der durch die Funktion zurueckgelieferte Wert undefiniert.

Das folgende Programm demonstriert saemtliche moeglichen Verwendungen von EXIT. EXIT-Befehle koennen in derselben Weise geschachtelt werden wie die jeweiligen Strukturen - innerhalb einer Prozedur kann beispielsweise ein SELECT-Befehl stehen, der eine oder mehrere Schleifen mit DO / LOOP enthaelt.

Beispiel:

Die ersten drei Verwendungsmoeglichkeiten:
Beenden von SELECT, IF / THEN / ELSE und der
Prozedur selbst.

```
SUB PROC(Wahl%,Zahl%)
   SELECT CASE Wahl%
      CASE 1
      'Pruefung mit SELECT
          SELECT CASE Zahl%
              CASE < 0
                 PRINT "Nummer ist kleiner als 0."
                 EXIT SELECT 'beendet CASE Zahl%
              CASE > 0
                 PRINT "Nummer ist groesser als 0."
                 EXIT SELECT 'beendet CASE Zahl%
              CASE ELSE
                 PRINT "Nummer ist 0."
           END SELECT
           EXIT SUB 'beendet PROC
      CASE 2
      'dieselbe Pruefung mit IF / THEN / ELSE
           IF Zahl% < 0 THEN
                 PRINT "Nummer ist kleiner als 0."
                 EXIT IF      'beendet IF
              ELSEIF Zahl% > 0 THEN
                 PRINT "Nummer ist groesser als 0."
                 EXIT IF      'beendet IF
              ELSE
                 PRINT "Nummer ist 0."
           END IF
   END SELECT
   PRINT "Sie haben '2' gewaehlt."
END SUB

'Die weiteren Verwendungsmoeglichkeiten
'von EXIT: Beenden von FOR / NEXT, DO / LOOP_
'und einer Funktionsdefinition.
```

```
'Die Funktion bekommt ueber einen Parameter
'mitgeteilt, welcher der EXIT-Befehle
'ausgefuehrt werden soll.

DEF FNKontrolle(Wahl%)
'der zurueckgelieferte Wert spielt in
'diesem Fall keine Rolle und kann deshalb
'sofort gesetzt werden:
FNKontrolle = -1

SELECT CASE Wahl%
     CASE 1
        'die folgende Schleife wird nur
        'einmal durchlaufen
        FOR i% = 1 TO 32767
          PRINT RND(Zahl%)
          IF i% = 1 THEN EXIT FOR
        NEXT i%
        EXIT DEF  'beendet die Funktion.
     CASE 2
        'auch DO / LOOP uebersteht ein EXIT nicht -
        'selbst dann, wenn es sich um eine
        'ansonsten endlose Schleife handelt
        DO
         PRINT RND(Zahl%)
         EXIT LOOP
        LOOP
        EXIT DEF     'beendet die Funktion.
     END SELECT
     PRINT "Sie haben die '2' gewaehlt."
END DEF         'fuer '2': normales Funktionsende

'********** Hauptprogramm ************

'drei Aufrufe von FNKontrolle
PRINT FNKontrolle(1)
PRINT FNKontrolle(2)
PRINT FNKontrolle(3)

INPUT "Geben Sie eine Zahl ein:"; N%
FOR Z% = 1 TO 2
     CALL PROC(Z%, N%)
NEXT Z%

END
```

4.2.8. Dimensionieren von Feldern

- DIM

Festlegen und Erzeugen von Feldern.

Syntax:

```
DIM STATIC|DYNAMIC Bezeichner(Index
    [,Index]...)[,Bezeichner(Index[,Index]...)]...
DIM STATIC|DYNAMIC Bezeichner(Bereich
    [,Bereich]...)[,Bezeichner
    (Bereich[,Bereich]...)]
```

Bemerkungen:

Dieser Befehl deklariert und erzeugt ein oder mehrere Felder mit angegebenem Typ und Namen sowie der angegebenen Anzahl und Groesse von Dimensionen.

Bezeichner steht fuer einen Variablennamen, der den Regeln von TBASIC entspricht (d.h. mit einem Buchstaben beginnt und danach eine beliebige Anzahl von Buchstaben und Ziffern enthaelt). Der Typ der Feld-Elemente wird entweder durch eine entsprechende Kennzeichnung des Feldnamens (mit %, & ...) oder durch eine vorherige DEF Typ-Anweisung festgelegt.

Index ist ein positiver Integer-Ausdruck und steht fuer die Groesse einer einzelnen Dimension. Ueber den Parameter Index wird die Nummer des hoechsten verfuegbaren Elements festgelegt. Der Befehl

```
DIM Nummer%(5)
```

erzeugt ein eindimensionales Feld mit dem Feldnamen Nummer%, das sechs Elemente vom Typ Integer enthaelt, die von 0 bis 5 durchnumeriert sind. Das erste Element wird mit Nummer%(0) bezeichnet, das zweite mit Nummer%(1) usw..

Die wiederholte Angabe von **Index** erzeugt ein Feld mit einer entsprechenden Anzahl von Dimensionen. Der Befehl

```
DIM Zeilen$(2,5)
```

erzeugt ein zweidimensionales Feld mit 18 Elementen vom Typ Zeichenkette. Das erste Element wird mit Zeilen$(0,0) bezeichnet, das zweite mit Zeilen$(0,1), das letzte mit Zeilen$(2,5).

Mit einem DIM - Befehl koennen mehrere Felder dimensioniert werden. Die beiden vorigen Beispiel-Felder lassen sich also auch folgendermassen erzeugen:

```
DIM Nummer%(5), Zeilen$(2,5)
```

Die zweite Form des Befehls DIM stellt eine Erweiterung der Sprachdefinition von BASIC dar. **Bereich** steht fuer zwei positive Integer-Ausdruecke, die durch einen Doppelpunkt (:) voneinander getrennt sind und die Index-Nummer des ersten bzw. letzten Elements einer Dimension bezeichnen. Der Befehl

```
DIM Beispiel(50:60)
```

erzeugt das eindimensionale Feld **Beispiel**, das elf Elemente vom Typ einfache Genauigkeit enthaelt. Das erste Element wird mit Beispiel(50), das letzte mit Beispiel(60) bezeichnet.

Ein Mischen beider Formen sowie das Erzeugen multidimensionaler Felder ist ebenfalls moeglich. Der Befehl

```
DIM Neu(80:100,4,1800:1899)
```

erzeugt ein dreidimensionales Feld, die Indexnummern der ersten Dimension gehen von 80 bis 100, die der zweiten von 0 bis 4, und die der dritten von 1800 bis 1899. Insgesamt enthaelt dieses Feld 21 * 5 * 100 = 10500 Elemente.

TBASIC setzt beim Start eines Programms saemtliche Elemente numerischer Felder auf den Wert 0, Zeichenketten-Feldelemente bekommen eine Null-Zeichenkette ("") zugewiesen. Ein Restart des Programms mit RUN sowie der Befehl CLEAR fuehren diese Initialisierung erneut aus.

Der Zugriff auf ein nicht vorher dimensioniertes Feld hat ein automatisches Erzeugen dieses Feldes zur Folge, wobei fuer jede angegebene Indexnummer eine Dimension mit 11 Elementen (maximaler Index: 10) erzeugt wird.

Z.B. haben die Befehle

```
x = N(4)
y = M(8,5)
```

die Dimensionierung eines ein- bzw. zweidimensionalen Feldes zur Folge, falls die angegebenen Feldvariablen nicht vorher explizit dimensioniert wurden. Eine Befehlsfolge wie

```
A = 11:Z = B(A)
```

erzeugt dagegen einen Laufzeitfehler, wenn B nicht zuvor dimensioniert wurde (die maximale Indexnummer automatisch erzeugter Felder ist 10). Dabei ist zu beachten, dass dieser Fehler nur dann gemeldet wird, wenn die Option **Bounds** im Menue **Options** vor dem Compilieren auf ON gesetzt ist. Es ist deshalb zu empfehlen, generell die Felder zu dimensionieren.

Ueber die (optionalen) Zusaetze **STATIC** und **DYNAMIC** wird festgelegt, ob der Compiler den Speicherplatz fuer ein Feld bereits waehrend des Uebersetzens belegt (STATIC) oder ob diese Belegung erst zur Laufzeit des Programms erfolgen soll. Die Standardvorgabe ist STATIC, sie kann ueber den Compiler-Befehl $DYNAMIC geaendert werden.

In zwei Faellen erzeugt der Compiler immer dynamische Felder:

- Bei der Festlegung einer oder mehrerer Dimensionsgroessen durch Variablen. Bei einer Befehlsfolge wie

  ```
  A = 20: DIM Feld(A)
  ```

  ist dem Compiler der Wert der Variablen A unbekannt, die statische Belegung einer bestimmten Speicherplatz-groesse also unmoeglich. Der Versuch, ein derartiges Feld mit dem Zusatz STATIC zu dimensionieren, erzeugt einen Fehler beim Compilieren.

- Beim Dimensionieren innerhalb von Funktionen und Prozeduren, falls das Feld nicht zuvor explizit als SHARED oder STATIC deklariert wurde.

  Zu beachten:

  Ein statisches -moeglichst noch globales- Feld sollte nie innerhalb einer Funktion oder Prozedur dimensioniert werden. Jeder weitere Aufruf dieser Definition haette den Laufzeitfehler 10 (doppelte Dimensionierung) zur Folge.

Dynamisch erzeugte Felder werden durch den Befehl ERASE komplett geloescht. Der belegte Speicherplatz steht danach fuer andere Zwecke zur Verfuegung. Das Anwenden von ERASE auf statisch erzeugte Felder setzt lediglich saemtliche Feld-Elemente zurueck und hat keine Freigabe von Speicherplatz zur Folge. Das Feld kann nach dem Loeschen weiterverwendet werden.

Grenzen:
Indexnummern sind grundsaetzlich positive Integerzahlen. Der zulaessige Bereich geht von 0 bis 32767. Die minimal moegliche Indexnummer kann mit der Anweisung OPTION BASE gesetzt werden. Ein Feld kann maximal 8 Dimensionen haben. Fuer jedes einzelne numerische Feld stehen maximal 64 KByte Speicherplatz zur Verfuegung.

Zeichenkettenfelder enthalten im Gegensatz zu numerischen Feldern die Werte der einzelnen Elemente nicht in direkter Form, sondern lediglich zwei Byte fuer die Laenge und einen Zeiger auf den jeweiligen Inhalt, der im Zeichenketten-Segment des Programms gespeichert ist. Ein Zeichenkettenelement kann eine maximale Laenge von 32767 Zeichen haben. Fuer saemtliche Zeichenkettenelemente und einfache Zeichenkettenvariablen stehen insgesamt 64 KByte Speicherplatz zur Verfuegung.

Ein Pruefen auf Ueberschreiten der Feldgrenzen durch falsches Indizieren wird nur dann vorgenommen, wenn der Compiler - Schalter **Bounds** im Menue **Options** zur Zeit des Compilierens auf ON gesetzt ist. In diesem Fall wird vor jedem Zugriff eine Pruefung vorgenommen, was die Ausfuehrungsgeschwindigkeit des Programms etwas herabsetzt.

Unterschiede:
TBASIC ermoeglicht das Verwenden von Bereichen anstelle von Obergrenzen beim Dimensionieren von Feldern.
BASI kann Felder nur in dynamischer Form verwalten. TBASIC bemueht sich dagegen, den entsprechenden Speicherplatz waehrend des Compilierens statisch zu belegen (Zugriffe auf statische Feld-Elemente werden erheblich schneller ausgefuehrt). Das Erzeugen statischer Felder kann entweder mit dem Compiler-Befehl $DYNAMIC oder dem Zusatz DYNAMIC innerhalb eines DIM-Befehls unterbunden werden.

Beispiel:

```
DIM A(20)               'konstante Groesse:statisch
Groesse = 40
DIM B(Groesse)          'variable Groesse:dynamisch
DIM DYNAMIC C(20)       'trotz Konstante:dynamisch
DIM STATIC X(Groesse)   'Fehler !!

ERASE A                 'setzt nur die Elemente von
                        'Feld A zurueck
ERASE B                 'loescht Feld B komplett
```

- OPTION BASE

Es wird eine Untergrenze fuer das Indizieren von Feldern festgelegt.

Syntax: **OPTION BASE Integer-Ausdruck**

Bemerkungen:

Integer-Ausdruck legt die niedrigste Indexnummer nachfolgender Feld-Dimensionierungen fest. Zulaessige Werte gehen von 0 bis 32767.

Auf OPTION BASE folgende DIM-Befehle erzeugen Felder, deren erstes Element die angegebene Nummer hat.
Ein Befehl wie

```
DIM A(20)
```

erzeugt normalerweise ein Feld mit 21 Elementen, die von 0 bis 20 numeriert sind. Wenn zuvor die Anweisung OPTION BASE 1 programmiert wurde, dann hat das Feld A nur 20 Elemente, die Nummer des ersten Elements ist 1.

Beispiel:

```
'ein Feld mit drei Integer-Elementen
DIM ErstesFeld%(2)

'Setzen des Start-Index auf 1
OPTION BASE 1

'und Erzeugen eines zweiten Feldes
'mit zwei Elementen
DIM ZweitesFeld%(2)

'Setzen der Feld-Inhalte: Beim zweiten
'Feld muss darauf geachtet werden, dass das
'Element 0 nicht existiert
FOR I% = 0 TO 2
     ErstesFeld%(I%)=I%
     IF I% > 0 THEN ZweitesFeld%(I%)=I%
NEXT I%

'und Ausgabe der Werte. Hier gilt fuer
'ZweitesFeld dasselbe:
FOR I% = 0 TO 2
     PRINT ErstesFeld%(I%)
     IF I% > 0 THEN PRINT ZweitesFeld%(I%)
NEXT I%

END
```

- ERASE

Loeschen dynamischer Felder bzw. Zuruecksetzen statischer Felder.

Syntax: **ERASE Feldname [,Feldname...]**

Bemerkungen:

Wenn **Feldname** ein dynamisches Feld bezeichnet, wird der durch dieses Feld belegte Speicherplatz wieder vollstaendig freigegeben. Der Feldname kann in einem spaeteren DIM-Befehl erneut verwendet werden.

Wenn **Feldname** ein statisches Feld bezeichnet, werden lediglich alle Elemente dieses Feldes auf den Wert 0 bzw. eine Nullzeichenkette zurueckgesetzt.

Auf statische Felder kann ERASE im Verlauf eines Programmes beliebig oft angewendet werden. Bei dynamischen Feldern wird zusammen mit den Felddaten auch der Namenseintrag geloescht - das erneute Anwenden von ERASE auf ein bereits geloeschtes Feld endet mit einem Laufzeitfehler, weil der Namenseintrag nicht mehr lokalisiert werden kann.

Unterschiede:
Statische Felder sind in BASI nicht definiert, ERASE entfernt bei BASI ein Feld immer komplett.

Beispiel:

```
ON ERROR GOTO Fehlerroutine  'Abfangen von Fehlern

'Ausgabe des freien Speicherplatzes fuer Felder
PRINT FRE(-1)

'Erzeugen eines dynamischen Feldes, Setzen der
'Elemente, erneute Ausgabe des freien Speichers
DIM DYNAMIC Feld(10000)
Feld(5000) = 100
PRINT FRE(-1)

'Loeschen und Ausgabe des freien Speicherplatzes
ERASE Feld
PRINT FRE(-1)

'Nachdem Feld geloescht ist, erzeugt der
'Zugriff darauf einen Laufzeitfehler - aber nur
'dann, wenn "Bounds" auf "ON" gesetzt ist. Ansonsten
'wuerde ein neues Feld mit 11 Elementen erzeugt,
'das danach unzulaessig indiziert wird:
PRINT Feld(5000)
END

Fehlerroutine:
     IF ERR = 9 THEN
          PRINT "Feld ist geloescht!"
     ELSE
          PRINT "Fehler: ";ERR
          PRINT "Adresse: ";ERADR
     END IF
END
```

4.2.9. Anwendereigene Definition einer Funktion / Prozedur

- DEF FN / END DEF

Definiert eine einzeilige (DEF FN) oder mehrzeilige Funktion (DEF FN / END DEF).

Syntax:

- Einzeilige Funktion:

```
DEF FNBezeichner[(Argumentenliste)]
     = Ausdruck
```

- Mehrzeilige Funktion:

```
DEF FNBezeichner[(Argumentenliste)]
[LOCAL Variablenliste]
[STATIC Variablenliste]
[SHARED Variablenliste]
     .
     .  Anweisungen
     .
     [EXIT DEF]
     [FNBezeichner = Ausdruck]
END DEF
```

Bemerkungen:

Bezeichner steht fuer den Namen der Funktion, welcher den Regeln von TBASIC entsprechen muss, d.h. grundsaetzlich mit einem Buchstaben beginnt und danach eine beliebige Anzahl von Buchstaben und Ziffern enthaelt. Mehrere Funktionen und/oder Prozeduren mit demselben Namen innerhalb eines Programms sind nicht gestattet und fuehren zu einer Fehlermeldung des Compilers.

Argumentenliste ist eine (optionale) Folge von formalen Parametern, die voneinander durch Kommas getrennt werden. In dieser Liste angegebene Namen haben eine rein formale Funktion und nichts mit eventuellen "globalen" Variablen gleichen Namens zu tun.

Die reservierten Woerter **DEF FN** und **END DEF** markieren den Beginn bzw. das Ende einer Folge von Befehlen, die unter dem angegebenen Namen als **Funktion** zusammengefasst werden. Eine Funktion kann bei ihrem Aufruf (optional) einen oder mehrere Parameter als Werte uebergeben bekommen und liefert grundsaetzlich einen Wert zurueck. Der Typ dieses Wertes wird (in derselben Weise wie bei Variablen) durch den Funktionsnamen festgelegt. Eine Funktion kann aus diesem Grund innerhalb beliebiger Befehlskonstrukte anstelle einer Variablen oder Konstanten entsprechenden Typs eingesetzt werden.

Funktionsdefinitionen und der Programmablauf:

Eine Funktiondefinition ist fuer ein ablaufendes Programm (in derselben Weise wie Prozeduren und DATA-Anweisungen) immer verfuegbar, egal, ob die Funktion in der ersten oder der tausendsten Zeile des Quelltextes definiert wurde. Im Gegensatz zu BASI ist ein "Durchlaufen" auch bei einzeiligen Funktionen nicht notwendig.

In Bezug auf die Programmausfuehrung kann eine Funktion als vom restlichen Code vollstaendig isoliert betrachtet werden. Ein versehentliches "Hineinfallen" in die Definition (wie bei Unterprogrammen) ist nicht moeglich.

Achtung:
Da Funktionsdefinitionen voellig vom restlichen Code des Programms isoliert sind, duerfen sie **nur ueber den Funktionsnamen** aufgerufen werden - ein Hinein -oder Herausspringen mit GOTO, GOSUB oder RETURN ist nicht moeglich. Innerhalb einer Funktionsdefinition sind diese Befehle allerdings erlaubt, solange dadurch die Definition nicht verlassen wird. Eine Funktionsdefinition kann also eine oder mehrere Ebenen von Unterprogrammen und/oder Spruenge mit GOTO enthalten.

Funktionsdefinitionen koennen nicht verschachtelt werden: Innerhalb einer Definition darf keine zweite Definition stehen. Aufrufe anderer Funktionen und Prozeduren **innerhalb** einer Definition sind dagegen legal.

Deklaration lokaler Variablen

Lokale Variablen koennen nur innerhalb von mehrzeiligen Funktionsdefinitionen deklariert werden. Eine oder mehrere Variablen werden durch das reservierte Wort LOCAL als lokal deklariert. Diese Deklaration muss **vor** dem ersten Verwenden der entsprechenden Variablen erfolgen. Mit LOCAL deklarierte Variablen sind nur temporaer zugeordnete Speicherplaetze, sie verlieren ihren Wert in dem Moment, in dem die Funktion verlassen wird. Eine Anweisung wie

```
LOCAL A%, B#, Zahlen%()
```

erzeugt drei lokale Variablen: die einfache Integervariable A%, die Variable doppelter Genauigkeit B# und das Integer-Feld Zahlen%. Dieses Feld muss nach seiner Deklaration entsprechend dimensioniert werden:

```
DIM DYNAMIC Zahlen%(1020)
```

Als LOCAL deklarierte Felder werden (wie alle anderen LOCAL-Variablen auch) beim Verlassen der Funktion/Prozedur wieder geloescht.

TBASIC verwaltet als LOCAL definierte Variablen wie folgt:

- Uebergebene Werte und Adressen werden auf dem Stack gespeichert.

- Als LOCAL deklarierte Variablen bekommen vom Compiler einen Speicherplatz im Datenbereich zugewiesen. Dieser Speicherplatz ist von dem globaler Variablen getrennt. Lokale Variablen sind unabhaengig von Variablen des restlichen Programms, belasten aber den Stack nicht.
  Vorteile: Der Zugriff ist erheblich schneller, die Groesse ist nicht auf 64 KByte begrenzt.

- Bei rekursiven Aufrufen einer Funktion werden die Werte lokaler Variablen (aus dem Datenbereich) gelesen und auf den Stack gebracht. **Nur in diesem Fall** setzt die Groesse des Stacks eine Grenze.

Die Attribute STATIC und SHARED

Als STATIC deklarierte Variablen sind rein lokal zur entsprechenden Funktions - oder Prozedurdefinition, behalten aber im Gegensatz zu mit LOCAL deklarierten Variablen ihren Wert zwischen einzelnen Aufrufen. Ihre Werte koennen nur innerhalb der jeweiligen Definition veraendert werden. Ein Speichern auf dem Stack findet nicht statt - auch nicht bei rekursiven Definitionen:

```
DEF FNVersuch$(L$)
STATIC X%
     PRINT FRE(-2)        'Platz auf dem Stack
     PRINT X%, L$
     INCR X%
     IF X% < 4 THEN FNVersuch$ = FNVersuch$
(L$ + "A")
END DEF

PRINT FNVersuch$("B")
```

Dieses Programm gibt die Zeichenfolgen B, BA, BAA und BAAA aus, L$ wird jeweils auf dem Stack uebergeben. Die Variable X% hat dagegen einen konstanten Speicherplatz und wird bei jeder Rekursion um eins erhoeht. Eine Deklaration von X% als LOCAL haette zur Folge, dass in jeder Rekursionsebene mit einem "neuen" X% gearbeitet wuerde und FNVersuch$ nie zu einem Ende kaeme.

Innerhalb einer Funktions- oder Prozedurdefinition wird jeder Variablen, die nicht zuvor explizit deklariert wurde, standardgemaess das Attribut STATIC zugeordnet.

Als SHARED deklarierte Variablen gelten sowohl innerhalb der Prozedur/Funktionsdefinition als auch fuer den Rest des Programms, sind also "global". Die Deklaration einer Variablen mit diesem Attribut stellt die einzige Moeglichkeit dar, ueber die eine Funktion Variablen des Hauptprogramms veraendern kann:

```
DEF FNVersuch%
LOCAL X%                          'dynamisch und lokal
STATIC Y%                         'statisch und lokal
SHARED Z%                         'statisch und global
     FNVersuch% = Z%              'ein Zugriff
     Z% = 1                       'und eine Veraenderung
     X% = 2: Y% = 3
END DEF

X% = 10: Y% = 11: Z% = 12         'globale Variablen
PRINT FNVersuch%                  'ergibt 12
PRINT X%, Y%                      'ergibt 10, 11
```

Funktionsergebnisse

Eine einzeilige Funktion enthaelt die Zuweisung eines Wertes zum Funktionsnamen. Innerhalb mehrzeiliger Funktionen ist dagegen eine explizite Zuweisung erforderlich. Wenn sie fehlt, ist der von der Funktion uebergebene Wert undefiniert.

Mit der Anweisung **EXIT DEF** kann eine Funktion "vorzeitig" verlassen werden - dieser Befehl kommt einem GOTO zu der Anweisung END DEF gleich.

- SUB, SUB INLINE und END SUB

Definieren einer Prozedur (Subprogramm).

Syntax:

```
SUB Bezeichner[(Parameterliste)] [INLINE]
[LOCAL Variablenliste]
[STATIC Variablenliste]
[SHARED Variablenliste]
     .
     .    Anweisung(en)
     .
     [EXIT SUB]
END SUB
```

Bemerkungen:

Bezeichner steht fuer den Namen der Prozedur, der den Regeln von TBASIC entsprechen muss. D.h., er muss mit einem Buchstaben beginnen, auf den eine beliebige Anzahl von Buchstaben und Ziffern folgen kann. Jeder Prozedurname darf innerhalb desselben Programms nicht gleichzeitig zur Bezeichnung anderer Prozeduren, Funktionen, Labels oder Variablen verwendet werden.

Parameterliste ist eine (optionale) Folge von Variablennamen, die voneinander durch Kommas getrennt sind und eine formale Funktion haben. Sie repraesentieren bei einem Aufruf der Prozedur uebergebene Werte und haben nichts mit eventuell gleichnamigen Variablen des Hauptprogramms zu tun. Parameter koennen einer Prozedur entweder ueber ihren Wert oder als Adresse uebergeben werden (siehe 3.6., "Prozeduren und Funktionen").

Die reservierten Woerter SUB und END SUB markieren den Beginn bzw. das Ende einer Folge von Befehlen, die unter dem angegebenen Namen als **Prozedur** zusammengefasst werden. Einer Prozedur koennen bei ihrem Aufruf (optional) ein oder mehrere Parameter uebergeben werden, sie liefert (im Gegensatz zu einer Funktion) keinen Wert zurueck.

INLINE-Prozeduren

Durch Anfuegen des reservierten Wortes INLINE nach der (optionalen) Parameterliste wird dem Compiler mitgeteilt, dass die entsprechende Prozedur $INLINE-Befehle (d.h. direkte Definitionen von Maschinenbefehlen fuer den Prozessor) enthaelt. Weitere Details siehe 4.10., Compiler-Befehl $INLINE.

Prozedurdefinitionen und der Programmablauf

Eine Prozedurdefinition ist fuer ein ablaufendes Programm in derselben Weise wie Funktionen und DATA-Anweisungen immer verfuegbar, egal, ob die Funktion in der ersten oder tausendsten Zeile des Quelltextes definiert wurde.

In Bezug auf die Programmausfuehrung kann eine Prozedur als vom restlichen Code vollstaendig isoliert betrachtet werden, ein versehentliches "Hineinfallen" in die Definition (wie bei Unterprogrammen) ist nicht moeglich.

Die Prozedur **ProcTest** wird im folgenden Beispiel nur einmal ausgefuehrt, naemlich durch den Aufruf in der ersten Zeile:

```
CALL ProcTest

SUB ProcTest
     PRINT "Hier ProcTest..."
END SUB
```

Achtung:
Prozedurdefinitionen duerfen **nur ueber den Prozedurnamen** aufgerufen werden - ein Hinein- oder Herausspringen mit GOTO, GOSUB oder RETURN ist nicht gestattet. Innerhalb einer Prozedurdefinition sind diese Befehle allerdings erlaubt, solange dadurch die Definition nicht verlassen wird. (Eine Definition kann also eine oder mehrere Ebenen von Unterprogrammen und/oder Spruenge mit GOTO enthalten.)

Prozedurdefinitionen koennen nicht verschachtelt werden. Innerhalb einer Definition darf keine zweite Definition stehen. **Aufrufe** anderer Funktionen und Prozeduren innerhalb einer Definition sind dagegen legal.
Mit dem Befehl EXIT SUB kann eine Prozedur "vorzeitig" verlassen werden. Er kommt einem GOTO zu der Anweisung END SUB gleich.

Uebergabe von Feldern als Parameter

Felder werden formal durch Anhaengen ihrer Dimensionsanzahl (nicht: Dimensionsgroesse) in Klammern an den Feldnamen gekennzeichnet. Die folgende Prozedur erwartet also ein zweidimensionales Feld und zwei Realzahlen einfacher Genauigkeit (die die jeweiligen Dimensionsgroessen angeben koennen - aber nicht muessen):

```
SUB Elementenzaehler(A(2), Anzahl1, Anzahl2)
```

Deklaration lokaler Variablen

Eine oder mehrere Variablen werden durch das reservierte Wort LOCAL als lokal deklariert. Diese Deklaration muss **vor** dem ersten Verwenden der entsprechenden Variablen erfolgen. Mit LOCAL deklarierte Variablen sind nur temporaer zugeordnete Speicherplaetze, sie verlieren ihren Wert in dem Moment, in dem die Funktion verlassen wird. Eine Anweisung wie

```
LOCAL A%, B#, Zahlen%()
```

erzeugt drei lokale Variablen: die einfache Integer-Variable A%, die doppeltgenaue Realvariable B# und das Integer-Feld Zahlen%.

Dieses Feld muss nach seiner Deklaration entsprechend dimensioniert werden:

```
DIM DYNAMIC Zahlen%(1000)
```

Alle als LOCAL deklarierten Felder werden (wie alle anderen LOCAL-Variablen auch) beim Verlassen der Funktion/Prozedur wieder geloescht.

TBASIC verwaltet als LOCAL definierte Variablen wie folgt:

- Beim Prozeduraufruf uebergebene Werte und Adressen werden auf dem Stack gespeichert.

- Als LOCAL deklarierte Variablen bekommen vom Compiler einen Speicherplatz im Datenbereich zugewiesen, der von dem globaler Variablen getrennt ist. Lokale Variablen sind unabhaengig von Variablen des restlichen Programms, belasten aber den Stack nicht.
  Vorteile: Der Zugriff ist erheblich schneller, die Groesse ist nicht auf 64 KByte begrenzt.

- Bei rekursiven Aufrufen einer Prozedur werden die Werte lokaler Variablen (aus dem Datenbereich) gelesen und auf den Stack gebracht. **Nur in diesem Fall** setzt das Fassungsvermoegen des Stacks eine Grenze.

Die Attribute STATIC und SHARED

Als **STATIC** deklarierte Variablen sind rein lokal zur entsprechenden Fuktions- und Prozedurdefinition, behalten aber im Gegensatz zu mit LOCAL deklarierten Variablen ihren Wert zwischen einzelnen Aufrufen. Ihre Werte koennen nur innerhalb der jeweiligen Definition veraendert werden. Ein Speichern auf dem Stack findet nicht statt, auch nicht bei rekursiven Definitionen:

```
SUB Versuch(V$)
STATIC X%
     PRINT "Stack: " FRE(-2)          'Platz auf dem Stack
     PRINT "Rekursionsebene: " X%
     INCR X%
     PRINT "Parameter: " V$
     IFX% < 4 THEN CALL Versuch(V$ + "a")
'Wert!
END SUB

CALL Versuch("x")
```

Dieses Programm gibt die Zeichenfolge x, xa, xaa und xaaa aus. Bei dem uebergebenen Parameter handelt es sich jeweils um einen Wert. Die Variable X% hat dagegen einen fixen Speicherplatz und wird bei jeder Rekursion um eins erhoeht. Eine Deklaration von X% als LOCAL haette zur Folge, dass in jeder Rekursionsebene mit einem "neuen" X% gearbeitet wuerde und die Prozedur "Versuch" nie zu Ende kaeme.

Innerhalb einer Prozedurdefinition wird standardmaessig jeder Variablen, die nicht zuvor explizit deklariert wurde, das Attribut STATIC zugeordnet.

Als **SHARED** deklarierte Variablen gelten sowohl innerhalb der Prozedur/Funktionsdefinition als auch fuer den Rest des Programms, sind also "global". Die Deklaration einer Variablen mit diesem Attribut stellt eine von zwei Moeglichkeiten dar, ueber die eine Prozedur Variablen des Haptprogramms veraendern kann (vgl. DEF FN):

```
SUB Versuch
LOCAL X%                        'dynamisch und lokal
STATIC Y%                       'statsich und lokal
SHARED Z%                       'statisch und global
     PRINT Z%                   'ein Zugriff
     Z% = 1                     'und eine Veraenderung
     X% = 2: Y% = 3
END SUB


X% = 10: Y% = 11: Z% = 12       'globale Variablen
CALL Versuch                    'gibt den Wert 12 aus
PRINT X%, Y%                    'ergibt 10,11 (unveraendert)
```

Zurueckliefern von Ergebnissen

Ueber die Parameter einer Prozedur ist die zweite Moeglichkeit gegeben, eine globale Variable zu veraendern und damit ein Ergebnis zurueckzuliefern. Je nachdem, ob ein Parameter als Wert oder als Adresse uebergeben wird (siehe Kapitel 3.6.), hat ein Veraendern des Parameters **innerhalb** der Prozedur eine Rueckwirkung oder findet rein lokal statt:

```
SUB Versuch(X%, Y%, Z%)
      INCR X%: INCR Y%: INCR Z%
END SUB

X% = 10: Y% = 20: Z% = 30
CALL Versuch(X%, (Y%), Z%+1)
PRINT X%, Y%, Z%                        'ergibt 11, 20, 30
```

In diesem Beispiel wird nur die Variable X% als Adresse uebergeben und deshalb direkt veraendert. Z%+1 ist ein Ausdruck. Y% ist durch Einschliessen in Klammern als Ausdruck "getarnt". Analoges gilt fuer "einfache" Variablen anderer Typen (inklusive Zeichenketten), Feld-Elemente koennen ebenfalls als Wert oder Adresse uebergeben werden, ganze Felder werden grundsaetzlich als Adresse behandelt.

Beispiel:

Die Prozedur erwartet folgende Parameter:
- ein eindimensionales Feld unbestimmter Groesse, das grundsaetzlich als Adresse uebergeben wird,
- zwei Integer-Werte oder Variablen
- einen Zeichenketten-Wert bzw. eine Zeichenketten-Variable

```
SUB ProcDemo (A(1),B%,C%,D$)
LOCAL B(), X                          'lokales Feld
SHARED CCC                            'eine globale Variable
STATIC DemoVar%                       'statisch

DIM B(5)                              'das lokale Feld

'Die Variable CCC ist "global", sie kann
'innerhalb von ProcDemo gelesen und / oder
'veraendert werden. B% ist der zweite
'uebergebene Parameter.
     PRINT CCC : CCC = CCC + B%

'DemoVar% ist STATIC, existiert also auch
'zwischen zwei Aufrufen von ProcDemo. X ist
'dagegen LOCAL, wird jedesmal neu erzeugt
'und auf den Wert 0 gesetzt:
     X = X + 1: PRINT X               'ergibt immer 1
     INCR DemoVar%
     PRINT DemoVar%                   'ergibt steigende Werte

'Das Feld B() wird bei jedem Aufruf neu
'erzeugt, seine Elemente haben den Wert 0:
     FOR B% = 0 TO 5
          PRINT B(B%),
     Next B%
     PRINT
     FOR B% = 0 TO 5                  'Setzen des lokalen Feldes
          B(B%) = B%                  'auf die Werte 0, 1, ..., 5
     NEXT B%
     PRINT B(4)                       'sollte "4" ergeben

'Das Feld A() ist als Adresse uebergeben worden,
'seine Elemente sind also veraenderbar:
     PRINT A(4)
     A(4) = A(4) + 1
```

```
'Ein Unterprogrammaufruf: TestSub
'liegt innerhalb der Prozedur !
     GOSUB TestSub

'Danach wird die Prozedur verlassen. Der
'EXIT-Befehl muss vor dem Unterprogramm
'stehen. Man kann auch innerhalb einer
'Prozedur in ein Unterprogramm "hineinfallen".
     EXIT SUB

TestSub:
'Je nachdem, ob C% und D$ als Werte oder als
'Adressen uebergeben worden sind, haben die
'folgenden Operationen eine Wirkung auf die
'Variablen (oder auch nicht):
     PRINT C%: C% = 0
     PRINT D$
     D$ = "Das  war eine Variable!"
     RETURN

END SUB


'*************   Hauptprogramm   *************

DIM X(20), Y(20)                    'zwei globale Felder

'Der erste Aufruf uebergibt das Feld X() als
'Adresse und die restlichen Parameter als Wert
CALL ProcDemo(X(), 5 ,4, "Zeichenketten-Konstante")

Parm$ = "Das ist eine Variable"
CCC = 100: Parm1% = 4:Parm2% = 3

'Der naechste Aufruf uebergibt die letzten drei
'Parameter ebenfalls als Adressen. Folge:
'Parm2 und Parm$ werden veraendert!
PRINT "Parm2 ist: ";Parm2%
PRINT "Parm$ ist: ";Parm$
CALL ProcDemo(Y(), Parm1%, Parm2%, Parm$)
PRINT "Nach dem Aufruf:"
PRINT "Parm2 ist: ";Parm2%
PRINT "Parm$ ist: ";Parm$

'Im folgenden Aufruf werden die letzten beiden
'Parameter wieder als Werte uebergeben, d.h.
'nicht veraendert. Grund: Parm2 ist als
'"Zwischenergebnis" getarnt, der vierte
'Parameter ist das Ergebnis einer Funktion:
Parm2% = 88: Parm$ = "Unveraendert."

CALL ProcDemo(Y(), Parm1%, (Parm2%), Parm$ + " ")
PRINT "Parm2 ist: ";Parm2%
PRINT "Parm$ ist: ";Parm$
```

```
'Der letzte Aufruf endet mit einem
'Laufzeitfehler. Grund: ProcDemo erwartet einen
'Integerwert als zweiten Parameter
'(automatische Konvertierung):

X! = 34756374              'zu hoch fuer "Integer"
'muss beim Aufruf als Ausdruck getarnt werden,
'sonst meldet sich der Compiler mit
'"unpassender Typ"
CALL ProcDemo(Y(), (X!), Parm2%, Parm$)

PRINT "Fehler"
END
```

- LOCAL

Deklaration von Variablen als "lokal" innerhalb einer Prozedur- oder Funktionsdefinition.

Syntax: **LOCAL Variablenliste**

Bemerkungen:

Variablenliste ist eine Folge von Bezeichnern, die durch Kommas voneinander getrennt sind.

Diese Anweisung kann nur innerhalb von Prozedur- und Funktionsdefinitionen benutzt werden. Sie muss innerhalb der Definition vor dem ersten auszufuehrenden Befehl stehen.

Als "lokal" deklarierte Variablen existieren fuer den Programmierer nur waehrend der Ausfuehrung der Prozedur bzw. Funktion.

Eine lokale Variable kann denselben Namen wie eine "globale" Variable des Hauptprogramms haben, die globale Variable bleibt von Veraenderungen der lokalen Variablen unberuehrt. Innerhalb voneinander getrennter Definitionen kann ein lokaler Variablenname mehrfach verwendet werden. Probleme ergeben sich auch dann nicht, wenn sich diese Definitionen gegenseitig aufrufen.

Im folgenden Beispiel existieren drei verschiedene Variablen mit dem Namen X, die jeweils vollstaendig voneinander getrennte Speicherplaetze des Compilers repraesentieren und sich deshalb gegenseitig nicht beeinflussen:

```
SUB Test1
LOCAL X
     X = 4
     CALL Test2
     PRINT X
END SUB
```

```
SUB Test2
LOCAL X
     X = 10
     PRINT X
END SUB

X = 18
CALL Test1
PRINT X
```

Um ein Feld als LOCAL zu deklarieren, muss der Feldname, gefolgt von einem leeren Klammernpaar, angegeben werden und auf die Deklaration ein entsprechender DIM-Befehl folgen. Lokale Felder muessen grundsaetzlich als DYNAMIC dimensioniert werden. Das geschieht entweder durch Verwenden einer Variablen zur Angabe der Dimensionsgroessen, durch den Zusatz DYNAMIC des DIM-Befehls oder durch Voranstellen des Compiler-Befehls $DYNAMIC.

Lokale Variablen werden bei jedem Aufruf der Prozedur/ Funktion auf den Wert 0 bzw. eine Null-Zeichenkette gesetzt.

Bei dem Speichern lokaler Variablen wird wie folgt verfahren:

- Waehrend des Compilierens wird der benoetigte Speicherplatz berechnet und innerhalb des Datenbereichs (statisch) belegt.

- Beim Aufruf einer Prozedur/Funktion werden die entsprechenden Speicherzellen geloescht, d.h. die lokalen Variablen auf den Wert 0 bzw. auf eine Null-Zeichenkette gesetzt.

- Wenn eine Prozedur oder Funktion rekursiv aufgerufen wird, dann liest TBASIC die Werte der dazugehoerigen lokalen Variablen und speichert sie auf dem Stack des Prozessors. Nach dem Aufruf werden die Werte wieder vom Stack heruntergeholt und zuruekgespeichert.

Damit wird ein Geschwindigkeitsvorteil erreicht, weil auch lokale Variablen direkt (d.h. nicht ueber den Stack) adressiert werden koennen. Ausserdem wird der Stack nur gering belastet. Aufgrund dieser Unterteilung koennen lokale Variablen soviel Platz belegen, wie Speicher zur Verfuegung steht. **Nur bei rekursiven Aufrufen** setzt die Groesse des Stacks eine Grenze.

Variablen, die innerhalb einer Prozedur oder Funktion ohne vorherige Deklaration benutzt werden, erhalten vom Compiler per Definition das Attribut **STATIC**. Variablen dieser Art sind lokal zur entsprechenden Definition, werden aber vor Aufrufen nicht geloescht bzw. bei Rekursion auf dem Stack gespeichert.

Es ist trotzdem zu empfehlen, grundsaetzlich saemtliche Variablen explizit zu deklarieren.

Grenzen:
Bei rekursiven Aufrufen wird der Stack des Prozessors benutzt zum Speichern von lokalen Variablen und uebergebenen Parametern. Das Stack-Segment des Prozessors bietet 32 KByte Platz, den sich saemtliche lokalen Variablen zusammen mit Parametern, Ruecksprungadressen usw. teilen muessen. Ein Ueberlauf des Stacks wird nur dann bemerkt, wenn vor dem Compilieren der Schalter **Stack test** im Menue **Options** auf ON gesetzt wurde.

Beispiel:

```
SUB LocalTest(X%)          'Parameter: implizit lokal
LOCAL A(), I%              'lokale Felder und Variable
SHARED Ergebnis%           'globale Variable
STATIC Zaehler%            'statische Variable

      DIM DYNAMIC A(10:20)
           FOR I% = 10 TO 20
                A(I%) = I%
                Ergebnis% = Ergebnis% + A(I%)
           NEXT I%
           INCR Zaehler%
           PRINT Zaehler% ".Aufruf zum LocalTest"
END SUB


'**********  Hauptprogramm  ********

PRINT "Ergebnis% = ";Ergebnis%          'ergibt 0
FOR I% = 1 TO 20                        'I% ist global!
     CALL LocalTest(I%)
     PRINT "Ergebnis% = ";Ergebnis%     'veraendert!
NEXT I%

PRINT Zaehler%                          'ergibt 0

END
```

- STATIC

Deklaration von Variablen innerhalb einer Funktions- oder Prozedurdefinition als statisch.

Syntax: **STATIC Variablenliste**

Bemerkungen:

Variablenliste ist eine Folge von Variablennamen, die voneinander durch Kommas getrennt sind. Felder werden innerhalb der **Variablenliste** durch Anfuegen eines leeren Klammerpaares gekennzeichnet. Die Reihenfolge der Variablennamen in der Liste spielt keine Rolle.

Als **STATIC** deklarierte Variablen sind (genauso wie als LOCAL deklarierte) nur innerhalb der jeweiligen Prozedur- oder Funktionsdefinition verfuegbar - sie haben mit den lokalen Variablen anderer Definitionen und "globalen" (SHARED) Variablen des Programms nichts zu tun.

Im Gegensatz zu als LOCAL deklarierten Variablen werden statische Variablen nur beim Start des Programms (sowie durch CLEAR oder RUN) auf den Wert 0 bzw. eine Null-Zeichenkette zurueckgesetzt. Sie behalten zugewiesene Werte zwischen den einzelnen Aufrufen der jeweiligen Prozedur/Funktion. Der Compiler reserviert einen festen Speicherplatz fuer sie. STATIC-Variablen bleiben auch bei rekursiven Aufrufen statisch, d.h. sie werden nicht auf dem Stack gespeichert.

Die Anweisung STATIC muss in der Definition vor dem ersten ausfuehrbaren Programmschritt stehen und kann nur innerhalb von Prozedur- und Funktionsdefinitionen verwendet werden.

Hinweis:

Nicht explizit deklarierten Variablen einer Prozedur ordnet der Compiler automatisch das Attribut STATIC zu.

Beispiel:

```
SUB Versuch
LOCAL A%                    'lokal (und dynamisch)
STATIC B%                   'lokal (und statisch)

'A% wird bei jedem Aufruf von Versuch
'erneut auf den Wert 0 gesetzt. B%
'dagegen nur beim Start des Programms.
'Der Beweis:

    INCR A%                 'ergibt jedesmal den Wert 1
    INCR B%                 'erhoeht sich jeweils

    PRINT "LOCAL-Variable: ";A%
    PRINT "STATIC-Variable: ";B%
END SUB


'**********  Hauptprogramm  ***********

FOR I% = 1 TO 20
    CALL Versuch
NEXT I%

END
```

- SHARED

Deklaration von Variablen innerhalb einer Prozedur- oder Funktionsdefinition als "global".

Syntax: **SHARED Variablenliste**

Bemerkungen:

Variablenliste ist eine Folge von Variablennamen, die durch Kommas voneinander getrennt sind. Felder werden in dieser Liste durch ein leeres Klammerpaar gekennzeichnet. Die Reihenfolge der Variablennamen in der Liste spielt keine Rolle.

Variablen werden durch **SHARED** innerhalb einer Funktions- oder Prozedurdefinition als "global" deklariert: Der Compiler erkennt daran, dass es sich um gemeinsame Variablen des Hauptprogramms **und** der jeweiligen Definition handelt. Diese Variablen koennen sowohl innerhalb der Prozedur bzw. Funktion als auch innerhalb des Hauptprogramms gelesen und/oder veraendert werden, im Gegensatz zu lokalen Variablen, die nur innerhalb der jeweiligen Definition gueltig sind. Als SHARED deklarierte Variablen haben grundsaetzlich statischen Charakter.

Die Anweisung **SHARED** darf nur innerhalb einer Prozedur- oder Funktionsdefinition verwendet werden und muss vor dem ersten Befehl stehen.

Hinweis:

Zuvor nicht deklarierte Variablen innerhalb einer Prozedur oder Funktion ordnet der Compiler das Attribut **SHARED** zu. Damit eine Prozedur/Funktion auf "globale" Variablen des Programms zurueckgreifen kann, muessen diese Variablen als SHARED deklariert werden.

Beispiel:

```
DEF FNTest$
SHARED Feld$()

    FNTest$ = Feld$(0)
    Feld$(1) = "und das auch."
END DEF
```

```
'************  Hauptprogramm  **********

DIM Feld$(5)        'ein globales Feld

Feld$(0) = "Das funktioniert"

PRINT FNTest$
PRINT Feld$(1)

END
```

4.2.10. Programmueberlagerung

- CHAIN

Eine weitere mit TBASIC erstellte Objektcode-Datei (Dateityp .TBC oder .EXE) wird in den Speicher geladen und die Programmausfuehrung mit dem geladenen Code fortgesetzt.

Syntax: **CHAIN Dateiname**

Bemerkungen: Wenn kein Dateityp angegeben ist, wird automatisch **.TBC** angefuegt.

Variablen, die in beiden Programmteilen (d.h. dem Programm, das die CHAIN-Anweisung enthaelt, und dem neu geladenen Programm) als **COMMON** deklariert sind, werden dem neu geladenen Programm unveraendert uebergeben. Falls sich Unstimmigkeiten zwischen beiden COMMON-Deklarationen ergeben (verschiedene Variablentypen, andere Reihenfolgen und/oder Feldgroessen), wird das Programm mit einem Laufzeitfehler abgebrochen.

Waehrend der Entwicklungs- und Testphase muss das aufrufende Programm als "**.EXE**"-Datei compiliert werden, weil CHAIN innerhalb der Programmierumgebung von TBASIC nicht moeglich ist. Das mit CHAIN zu ladende Programm kann dagegen sowohl als ".EXE" als auch als ".TBC"-Datei erzeugt werden.

CHAIN kann nur ausgefuehrt werden, wenn der erste der beiden Programmteile von DCP aus gestartet worden ist.

Unterschiede:
MERGE, DELETE, ALL und die Angabe einer Zeilennummer zum Start des neu geladenen Programms ab einer bestimmten Zeile werden bei TBASIC nicht unterstuetzt.

Das Abarbeiten eines neu geladenen Programmteils beginnt deshalb immer mit der ersten Zeile. Da TBASIC den zur Verfuegung stehenden Speicher aber wesentlich effektiver aus-nutzt, wird es in den meisten Faellen moeglich sein, mehrere ansonsten durch CHAIN verbundene Programmteile zu einem Programm zusammenzufassen - eine Verkettung mit CHAIN wird dabei ueberfluessig.

Beispiel:

```
'PROG1.EXE ist das Hauptprogramm
'und wird durch CHAIN mit PROG2 verbunden.

'Zuerst einige Variablen:
DIM C%(3000)
A=165 : B=12 : C%(2100)=120

'Deklaration als COMMON: fuer Felder wird
'nicht die Groesse, sondern die Dimensionsanzahl
'angegeben:
COMMON A, B, C%()
'und weiter mit PROG2
CHAIN "PROG2"

'PROG2.TBC  ist das nachgeladene Programm und wird von
'PROG1 gestartet. Zuerst muessen die als COMMON
'uebergebenen Variablen deklariert werden - und
'zwar mit demselben Typ und in derselben
'Reihenfolge.
'Die Namen spielen dagegen keine Rolle.
COMMON X, Y, Z%()
PRINT X, Y, Z%(20.00)
```

PROG1.EXE muss von DCP aus gestartet werden. Nach Ausfuehrung der angegebenen Befehle laedt es PROG2.TBC in den Speicher (und ueberschreibt sich damit selbst). PROG2 enthaelt in unserem Beispiel keine weitere CHAIN-Anweisung und endet mit einem Ruecksprung in die Kommandoebene von DCP.

- COMMON

Uebergibt eine odere mehrere Variablen zwischen zwei verketteten Programmen.

Syntax: **COMMON Variable[,Variable...]**

Bemerkungen:

Ein mit CHAIN geladenes Programm muss entsprechende COMMON-Anweisungen enthalten, wobei die Namen der Variablen keine Rolle spielen, die Reihefolge der Deklaration und die jeweiligen Variablentypen aber strengstens eingehalten werden muessen. Jede Unstimmigkeit fuehrt zu einem Laufzeitfehler.

Unterschiede:
TBASIC unterstuetzt nicht CHAIN ALL des BASIC-Interpreters BASI. Jede zu uebergebende Variable muss mit COMMON uebergeben werden.
Bei der Uebergabe von Feldvariablen wird nicht die Groesse, sondern die Anzahl der Dimensionen in Klammern angegeben.

Bei BASI enthaelt nur der aufrufende Programmteil eine Deklaration mit COMMON, der neu geladene Programmteil dagegen nicht. Eine Pruefung, ob uebergebene Variablen im neu geladenen Programmteil korrekt behandelt werden, findet statt (jeder Variableneintrag enthaelt zusaetzlich den Variablennamen). Variableneintraege von TBASIC sparen den entsprechenden Speicherplatz - dafuer muessen aber Reihenfolge und Typ uebergebener Variablen exakt uebereinstimmen.

Beispiel:

Die beiden folgenden Programmteile demonstrieren die Uebergabe von Variablenwerten mit COMMON und CHAIN. Das erste Programm muss als ".EXE"-Datei, das zweite Programm als ".TBC"-Datei mit dem Namen **PROG2** compiliert werden. Um beide Programmteile auszuprobieren, muessen die Programmierumgebung von TBASIC verlassen und das erste Programm von der DCP-Kommandoebene gestartet werden.

```
'Programm 1 - muss als .EXE compiliert werden

DIM AltFeld$(7)           'ein Feld mit 8 Zeichenketten
AltFeld$(0) = "Erste Zeichenkette im Feld"
AltFeld$(1) = "Zweite Zeichenkette im Feld"

AltInteger% = 12345
AltLangInt& = 123450000
AltEinfachReal! = 12345.12345
AltDoppeltReal# = 1.0E+300
N$ = "Nicht uebergebene Zeichenkette."

'Deklaration der zu uebergebenden Variablen
COMMON AltFeld$(1), AltInteger%, AltLangInt&, _
       AltEinfachReal!, AltDoppeltReal#

CHAIN "PROG2"

'PROG2 - muss als ".TBC"-Datei compiliert werden

COMMON NeuFeld$(1), NeuInteger%, NeuLangInt&, _
       NeuEinfachReal!, NeuDoppeltReal#

PRINT NeuFeld$(0), NeuFeld$(1)
PRINT NeuInteger%, NeuLangInt&, _
      NeuEinfachReal!, NeuDoppeltReal#

PRINT "Hier folgt kein weiterer Text ->";
PRINT N$

END
```

4.2.11. Fehlerbehandlung

- ERROR

Gezieltes Erzeugen eines Lauzzeitfehlers.

Syntax: **ERROR n**

Bemerkungen:

n steht fuer die Nummer des Fehlers, dessen Auftreten simuliert werden soll.

Falls ON ERROR GOTO nicht gesetzt ist, endet das Programm mit der Meldung

Fehler n, pgm-ctr: Adresse.

n gibt die mit ERROR angegebene Fehlernummer an, **Adresse** den Stand des Programmzaehlers zum Zeitpunkt des ERROR-Befehls.

- ERR und ERL

Bestimmt den Fehlercode und die Zeilennummer, in der der Laufzeitfehler aufgetreten ist.

Syntax: **v = ERR**
v = ERL

Bemerkungen:

ERR uebergibt die Nummer des zuletzt aufgetretenen Fehlers im Integerformat.

ERL uebergibt die Zeilennummer des zuletzt aufgetretenen Fehlers als Integerzahl. Wenn der Fehler in einer Zeile auftrat, die im Quelltext ohne Zeilennummer geschrieben wurde, dann sucht ERL in Richtung auf den Beginn des Programms (bzw. des Quelltextes) und uebergibt die naechste gefundene Zeilennummer. Befindet sich zwischen der Fehlerstelle und dem Programmanfang keine numerische Zeile, liefert ERL den Wert 0.

Beide Funktionen sind ausschliesslich fuer die Behandlung von Fehlern in einer entsprechenden Routine vorgesehen und haben nur einen Sinn, wenn eine entsprechende Verzweigung mit ON ERROR GOTO gesetzt ist.

Hinweis:

Der Einsatz von **ERL**, **RESUME** und/oder **RESUME NEXT** veranlasst den Compiler, fuer **jeden Befehl** innerhalb eines Programms einen zusaetzlichen Zeiger (4 Byte) zu erzeugen - ein Befehl wie **RESUME (Zeilennummer)** besteht dagegen nur aus einem einzigen Sprungbefehl fuer den Prozessor. Falls bei sehr grossen Programmen Speicherprobleme auftreten, sollte auf die drei zuerst erwaehnten Befehle verzichtet werden.

Unterschiede:
TBASIC kommt ohne einen Befehl wie RENUM(ber) aus, deshalb gibt es bei Pruefungen von ERL keine Probleme: Vergleiche wie IF ERL <> 15 und IF 15 <> ERL werden vom Compiler als identisch behandelt. Spezielle Werte fuer ERL bei Fehlern im Direktmodus sind nicht definiert - schliesslich gibt es diesen Modus in TBASIC nicht.

- ON ERROR GOTO

Festlegen einer Routine zur Behandlung von Laufzeitfehlern.

Syntax: **ON ERROR GOTO Label**

Bemerkungen:

Wenn **ON ERROR GOTO** nicht gesetzt (oder wieder zurueckgesetzt) ist, wird die Programmausfuehrung von .EXE-Dateien bei einem Fehler mit der folgenden Meldung abgebrochen:

```
Fehler xxx, pgm-ctr = yyyy.
```

xxx steht fuer die Art des Fehlers, **yyyy** fuer den Stand des Programmzaehlers zum Zeitpunkt des Abbruchs. Wurde das Programm innerhalb der Programmierumgebung von TBASIC gestartet, dann wird automatisch der Editor aktiviert und der Kursor auf die Fehlerstelle im Quelltext gesetzt. In der obersten Zeile des Fensters **Edit** wird eine Fehlermeldung im Klartext angezeigt.

Nach einer **ON ERROR GOTO**-Anweisung bricht TBASIC die Ausfuehrung des Programms bei einem Fehler nicht mehr mit der oben genannten Fehlermeldung ab, sondern fuehrt einen Sprung zu dem Programmteil aus, der durch **Label** bezeichnet ist und die eigene Fehlerbehandlungsroutine beinhaltet.

Innerhalb der Fehlerbehandlungsroutine gibt es mehrere Moeglichkeiten: Ueber die Funktionen ERR und ERL kann die Fehlerstelle lokalisiert werden, RESUME (mit verschiedenen Zusaetzen) setzt das Programm auf definierte Weise fort.

Wenn **Label** in einer ON ERROR GOTO-Anweisung nicht angegeben ist oder hier die Zeilennummer 0 steht, dann benutzt TBASIC von diesem Zeitpunkt an wieder die Standard-Fehlerbehandlung von TBASIC.

ON ERROR GOTO kann innerhalb eines Programms beliebig oft gesetzt und zurueckgesetzt werden.

- ERADR

Uebergibt die Adresse des zuletzt aufgetretenen Fehlers.

Syntax: **v = ERADR**

Bemerkungen:

ERADR (fuer "ERROR Address" = Fehleradresse) uebergibt einen Langinteger, der dem Stand des Programmzaehlers beim zuletzt aufgetretenen Fehler entspricht. Dieser Wert ist exakt derselbe, der bei nicht gesetztem ON ERROR GOTO als **pgm-ctr** ausgegeben wird. Er kann unveraendert zur Bestimmung der Fehlerstelle im Quelltext eingegeben werden (Menue **Debug**, Wahlmoeglichkeit **Runtime error**).

Die Funktion **ERADR** wird innerhalb selbstgeschriebener Fehlerbehandlungsroutinen dann notwendig, wenn ein Fehler auftritt, fuer den keine Fehlerbehandlungsmassnahmen vorgesehen sind. In diesem Falle sollte die Routine zumindest die Nummer (siehe Funktion ERR) und die Stelle des Fehlers ausgeben, um eine spaetere Suche nach der Fehlerstelle zu ermoeglichen.

Beispiel:

```
'Sprung zur Fehlerbehandlungsroutine einleiten
ON ERROR GOTO Fehler

DIM X%(10)

FOR I% = 1 TO 10          'hier werden 10 Elemente
     READ X%(I%)          'gelesen, obwohl nur 7
NEXT I%                   'DATAs vorhanden sind

'7 DATAs. Der achte READ-Befehl erzeugt
'den Fehler "Keine weiteren DATA-Elemente".
'Da ON ERROR GOTO gesetzt ist, wird zur
'Routine "Fehler" gesprungen.
DATA 1, 2, 3, 4, 5, 6, 7

END                       'Programmende, nie erreicht


Fehler:
'Hier koennte eine Reihe von Pruefungen
'fuer erwartete Fehler stehen...
```

```
'"Unerwarteter Fehler": Ausgabe der Fehlernummer
'und des Programmzaehlerstandes.
PRINT: BEEP 2
PRINT "Fehler "; ERR
PRINT "Adresse "; ERADR
:
:
END
```

- RESUME

Setzt die Programmausfuehrung nach einer Fehlerbehandlung fort.

Syntax: **RESUME [0/NEXT/Label]**

Bemerkungen:

RESUME Label bewirkt, dass das Programm an der durch Label bezeichneten Stelle fortgesetzt wird.

Die Verwendung der Befehle **ERL**, **RESUME**, **RESUME 0** und **RESUME NEXT** veranlassen den Compiler, **fuer jeden Programmschritt** einen Zeiger (4 Byte) zu erzeugen, der beim Compilieren dem Datenbereich zugeschlagen wird. **RESUME Label** kommt dagegen ohne Zeiger aus: Der Compiler berechnet waehrend der Uebersetzung die Adresse von **Label** und generiert einen einzigen Sprungbefehl fuer den Prozessor (4 Byte).

4.2.12. Unterbrechungsabfrage

- KEY(n)

Aktiviert bzw. inaktiviert die Unterbrechung ("Abfangen") einer zuvor definierten Tasten (-Kombination).

Syntax: **KEY(n) ON/OFF/STOP**

Bemerkungen:

n ist ein Integer-Ausdruck im Bereich von 1 bis 31 und steht fuer eine der folgenden Tasten:

n	Taste
1 - 10	Funktionstasten F1 bis F10
11	Kursor nach oben
12	Kursor nach links
13	Kursor nach rechts
14	Kursor nach unten
15 - 29	mit KEY definierte Tasten (-Kombinationen)
30	Funktionstaste F11
31	Funktionstaste F12

KEY(n) STOP schaltet das "Abfangen" der entsprechenden Taste zeitweilig ab, speichert eventuelle Tastendruecke aber zwischen: Der naechste ON-Befehl fuer diese Taste hat einen sofortigen Aufruf des entsprechenden Unterprogramms zur Folge, wenn die Taste in der Zwischenzeit gedrueckt wurde.

- ON KEY(n)

Festlegen einer "Abfangroutine" (Unterbrechung) fuer eine bestimmte Tastenkombination.

Syntax: **ON KEY(n) GOSUB Label**

Bemerkungen:

n ist ein Integer-Ausdruck im Bereich von 1 bis 31 und legt fest, auf welchen Tastendruck hin das durch **Label** bezeichnete Unterprogramm aufgerufen werden soll. Die Zuordnung von **n** und einer Taste bzw. Tastenkombination ist wie unter Anweisung KEY(n) festgelegt.

Mit ON KEY wird lediglich der Zusammenhang zwischen einer "Tastennummer" und einem Unterprogramm hergestellt. Eine Reaktion von TBASIC findet erst dann statt, wenn die betreffende Taste mit dem Befehl KEY(n) ON aktiviert worden ist. Nach diesem Befehl prueft TBASIC vor Ausfuehren jedes Programmschrittes, ob diese Taste gedrueckt wurde. Wenn ja, wird das mit ON KEY(n) festgelegte Unterprogramm aufgerufen. Mit KEY(n) OFF wird diese Reaktion wieder abgeschaltet.

Mit dem Compiler-Befehl $EVENT kann kontrolliert werden, welche Teile eines Programms durch Tastendruecke und ON KEY unterbrechbar sind.

- ON TIMER

Legt ein Unterprogramm fest, das in regelmaessigen Abstaenden aufgerufen werden soll.

Syntax: **ON TIMER(n) GOSUB Label**

Bemerkungen:

n ist ein Integer-Ausdruck im Bereich von 1 bis 86400 und legt fest, wieviele Sekunden zwischen zwei Aufrufen des durch **Label** bezeichneten Unterprogramms vergehen sollen. Fuer n = 1 wird das Unterprogramm einmal pro Sekunde, fuer n = 86400 einmal in 24 Stunden aufgerufen.

Wie alle anderen ON... -Anweisungen legt auch **ON TIMER** lediglich eine Zuordnung fest. Eine sichtbare Reaktion von TBASIC auf **ON TIMER** erfolgt erst, nachdem der Befehl **TIMER ON** gegeben wurde. Nach diesem Befehl prueft TBASIC vor der Ausfuehrung jedes Programmschrittes, ob die angegebene Anzahl von Sekunden vergangen ist. Wenn ja, wird das durch **Label** bezeichnete Unterprogramm aufgerufen und der Sekundenzaehler zurueckgesetzt. Mit dem Befehl **TIMER OFF** laesst sich diese Art wieder abschalten.

Ueber den Compiler-Befehl $EVENT kann kontrolliert werden, welche Teile des Programms durch ein Zeitzaehlen unterbrochen werden.

Beispiel:

```
'Aufruf des Unterprogramms Zeitausgabe
'auf einmal pro Sekunde festgelegt
ON TIMER(1) GOSUB Zeitausgabe

'und das "Abfangen" aktivieren:
TIMER ON

'Das "Hauptprogramm": Es wartet nur
'auf einen Tastendruck, waehrend oben
'links im Bildschirm die Uhrzeit
'angezeigt wird.

LOCATE 4,1
PRINT "Bitte druecken Sie eine Taste:";
WHILE NOT INSTANT: WEND

END                                    'Programmende
```

```
Zeitausgabe:
'wird einmal pro Sekunde ausgefuehrt und laesst
'die originale Kursorposition unveraendert.
     K% = POS(K%): C% = CSRLIN   'Kursorposition
     LOCATE 1,1                  'neu setzen
     PRINT TIME$                 'Uhrzeit-Ausgabe
     LOCATE C%, K%               'und Kursor wieder setzen

RETURN
```

- ON PLAY

siehe Kapitel 4.7.2.

- COM(n) und ON COM(n)

siehe Kapitel 4.5.2.

4.3. BASIC-Grundfunktionen

4.3.1. Numerische Funktionen

4.3.1.1. Arithmetische Funktionen

- ABS

Uebergibt den absoluten Wert des Ausdruckes x.

Syntax: v = ABS(x)

Bemerkungen:
keine

- INT

Uebergibt den ganzzahligen Teil des Ausdruckes x.

Syntax: v = INT(x)

Bemerkungen:
keine

- FIX

Konvertiert x in einen ganzzahligen Wert durch Abschneiden der Nachkommastellen.

Syntax: v = FIX(x)

Bemerkungen:
keine

- CEIL

Uebergibt den naechstgroesseren ganzzahligen Wert oder das Argument selbst.

Syntax: v = CEIL(n)

Bemerkungen:

Im Gegensatz zur Funktion **INT** liefert **CEIL** nicht den ganzzahligen Anteil von n, sondern grundsaetzlich wird aufgerundet in Richtung hoeherer Werte. **CEIL(2.01)** ergibt den Wert 3. **CEIL (-2.8)** ergibt den Wert -2. Nur wenn das Argument keine Nachkommastellen hat, wird sein Wert unveraendert uebergeben.

Unterschiede:
Diese Funktion ist in BASI nicht vorhanden.

Beispiel:

- Vergleich zwischen Argument und den Funktionen CEIL und INT:

```
FOR I = -4.5 TO 4.5 STEP 0.25
  PRINT I, CEIL(I), INT(I)
NEXT I
```

- SGN

Ermittelt das Vorzeichen von x.

Syntax: **v = SGN(x)**

Bemerkungen:
keine

4.3.1.2. Konvertierung von Zahlenwerten

- CINT

Wandelt x in eine ganze Zahl um.

Syntax: **v = CINT(x)**

Bemerkungen:
keine

Unterschiede:
Im Gegensatz zu BASI rundet TBASIC nicht grundsaetzlich auf. Wenn der Wert nach dem Komma kleiner oder gleich 0.5 ist, wird abgerundet.

- CSNG

Wandelt x in eine Zahl einfacher Genauigkeit um.

Syntax: **v = CSNG(x)**

Bemerkungen:

Bei Zwischenergebnissen von Integeroperationen kann **CSNG** benutzt werden, evtl. Ueberlaeufe zu verhindern.

Beispiel:

```
Ein Ueberlauf: Das Zwischenergebnis von X%*Y% ist 40000
X% = 2000:Y% = 2000:Z% = 600
A = (X%*Y%) / Z%: PRINT A

und wie man ihn verhindert:
A = (CSNG(X%) * Y%) / Z%: PRINT A
```

- CDBL

Wandelt x in eine Zahl doppelter Genauigkeit um.

Syntax: `v = CDBL(x)`

Bemerkungen:

CDBL wird hauptsaechlich dazu angewendet, den Ablauf einer Rechnung in das doppeltgenaue Format zu zwingen, um damit evtl. Ueberlauf bei Zwischenergebnissen vorzubeugen.

Beispiel:

```
Y# = (X% + Z%) / A#
```

Obwohl das Ergebnis einer Variablen doppelter Genauigkeit zugewiesen wird, kann bei der Auswertung ein Ueberlauf stattfinden - naemlich dann, wenn X% + Z% einen Wert groesser 32767 ergeben. Die Anwendung von **CDBL** auf einen der beiden Operanden erzwingt die Ausfuehrung im Format doppelter Genauigkeit und bewahrt so das Programm vor Fehlern.

```
Y# = (CDBL(X%) + Z%) / A#
```

- CLNG

Wandelt x in das Format Langinteger.

Syntax: `v = CLNG(x)`

Bemerkungen:

Kann einen beliebigen Wert innerhalb des gueltigen Bereichs fuer Langinteger (-2 hoch 31 bis +2 hoch 31-1 oder rund +/-2 Milliarden) haben und wird (durch Rundung eventueller Nachkommastellen) in das Format Langinteger gebracht.

Grenzen:
Liegt der Wert ausserhalb des Gueltigkeitsbereichs fuer Langinteger, dann erzeugt **CLNG** den Laufzeitfehler 6 (Ueberlauf).

Unterschiede:
Diese Funktion ist (wie der Variablentyp Langinteger selbst) in BASI nicht definiert.

Beispiel:

```
PRINT CLNG(2.0E9),CLNG(33.4)
```

4.3.1.3. Exponentialfunktionen

- EXP

Errechnet e hoch x.

Syntax: **v = EXP(x)**

Bemerkungen:

x steht fuer einen numerischen Ausdruck, der sowohl ein Integerwert als auch eine Zahl einfacher oder doppelter Genauigkeit sein kann. Zulaessige Werte von x liegen im Bereich von +/-708. Kleinere Argumente unterschreiten den Gueltigkeitsbereich doppeltgenauer Realzahlen und liefern das Ergebnis Ø. Groessere Argumente ueberschreiten ihn (e hoch 708 entspricht dem Wert 1.0E+308).

EXP uebergibt das Ergebnis der Berechnung e^x in doppelter Genauigkeit, wobei e fuer die Basis des natuerlichen Logarithmus steht. e hat den Wert 2.71818282...

- EXP2

Errechnet 2 hoch x.

Syntax: **v = EXP2(x)**

Bemerkungen:

x steht fuer einen numerischen Ausdruck, der sowohl ein Integerwert als auch eine Zahl einfacher oder doppelter Genauigkeit sein kann. Zulaessige Werte von x liegen im Bereich von +/-1026. Kleinere Argumente unterschreiten den Gueltigkeitsbereich doppeltgenauer Realzahlen und liefern das Ergebnis Ø, groessere ueberschreiten ihn.

EXP2 liefert das Ergebnis der Berechnung 2^x in doppelter Genauigkeit. Die folgenden beiden Zeilen liefern dasselbe Ergebnis - die zweite wird allerdings um einiges schneller ausgefuehrt:

```
PRINT 2^68.25
PRINT EXP2(68.25)
```

- EXP10

Errechnet 10 hoch x.

Syntax: `v = EXP10(x)`

Bemerkungen:

x steht fuer einen numerischen Ausdruck, der sowohl ein Integerwert als auch eine Zahl einfacher oder doppelter Genauigkeit sein kann. Zulaessige Werte von x liegen im Bereich von +/-308. Kleinere Argumente unterschreiten den Gueltigkeitsbereich doppeltgenauer Realzahlen und liefern das Ergebnis Ø, groessere Argumente ueberschreiten ihn.

EXP10 liefert das Ergebnis der Berechnung 10^x in doppelter Genauigkeit. Die folgenden beiden Zeilen liefern dasselbe Ergebnis - die zweite wird allerdings um einiges schneller ausgefuehrt:

```
PRINT 10^68.25
PRINT EXP10(68.25)
```

- LOG

Uebergibt den natuerlichen Logarithmus (Basis e) von x.

Syntax: `v = LOG(x)`

Bemerkungen:

Als natuerlicher Logarithmus einer Zahl wird der Wert bezeichnet, mit dem e (2.718282) potenziert werden muss, um die Zahl als Ergebnis zu erhalten. **LOG(10)** ergibt 2.3025...,e^2.3025 ergibt wieder 10.

Der Logarithmus ist mathematisch nur fuer positive Zahlen definiert. Wenn x einen Wert kleiner oder gleich Ø hat, erzeugt **LOG** den Laufzeitfehler 5 (Illegaler Funktionsaufruf).

LOG uebergibt eine Zahl doppelter Genauigkeit.

- LOG2

Uebergibt den Logarithmus von x zur Basis 2.

Syntax: `v = LOG2(x)`

Bemerkungen:

Als Logarithmus einer Zahl wird der Wert bezeichnet, womit die Basis (also hier 2) potenziert werden muss, um die Zahl als Ergebnis zu erhalten. **EXP2(8)** ergibt 256, **LOG2(256)** ergibt wieder 8.

Der Logarithmus ist mathematisch nur fuer positive Zahlen definiert. Wenn x einen Wert kleiner oder gleich Ø hat, erzeugt **LOG2** den Laufzeitfehler 5 (Illegaler Funktionsaufruf).

LOG2 uebergibt eine Zahl doppelter Genauigkeit.

- LOG10

Uebergibt den Logarithmus von x zur Basis 10.

Syntax: **v = LOG10(x)**

Bemerkungen:

Als Logarithmus einer Zahl wird der Wert bezeichnet, womit die Basis (also hier 10) potenziert werden muss, um die Zahl als Ergebnis zu erhalten. **EXP10(4)** ergibt **1000**, **LOG10(1000)** ergibt wieder 4.

Der Logarithmus ist mathematisch nur fuer positive Zahlen definiert. Wenn x einen Wert kleiner oder gleich Ø hat, erzeugt **LOG10** den Laufzeitfehler 5 (Illegaler Funktionsaufruf).

LOG10 uebergibt eine Zahl doppelter Genauigkeit.

- SQR

Uebergibt die Quadratwurzel von x.

Syntax: **v = SQR(x)**

Bemerkungen:

Die Quadratwurzel einer negativen Zahl ist (in der Menge der reellen Zahlen) mathematisch nicht definiert. x muss also einen Wert haben, der groesser oder gleich Ø ist. Negative Argumente haben einen Laufzeitfehler 5 (Illegaler Funktionsaufruf) zur Folge.

SQR uebergibt eine Zahl doppelter Genauigkeit.

4.3.1.4. Trigonometrische Funktionen

- SIN

Errechnet den Sinus von x.

Syntax: `v = SIN(x)`

Bemerkungen:

x wird von der Funktion **SIN** als Angabe eines Winkels in der Einheit **Rad** (Wert von Ø bis 2 Pi) interpretiert. Um den Sinus aus einer Angabe in der Einheit Grad (Wert von Ø bis 360) zu berechnen, muss die folgende Formel benutzt werden:

```
y = SIN(GradWert / 57.2958)
```

Die Funktion **ATN** in diesem Kapitel beschaeftigt sich ausfuehrlicher mit Umrechnungen zwischen beiden Einheiten.

Die Funktion **SIN** uebergibt eine Zahl doppelter Genauigkeit im Bereich von -1 bis +1.
Sh. Funktion **ATN**.

- COS

Uebergibt den Cosinus von x.

Syntax: `v = COS(x)`

Bemerkungen:

COS uebergibt eine Zahl doppelter Genauigkeit im Bereich -1 bis +1.
Sh. Funktion **ATN**.

- TAN

Uebergibt den Tangens von x.

Syntax: `v = TAN(x)`

Bemerkungen:

TAN uebergibt eine Zahl doppelter Genauigkeit.
Sh. Funktion **ATN**.

- ATN

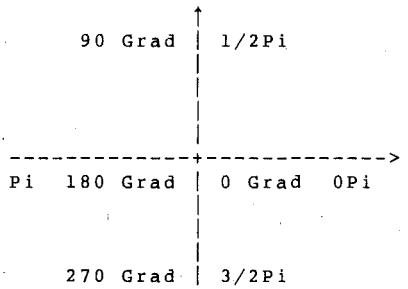
Uebergibt den Arcustangens von x.

Syntax: **v = ATN(x)**

Bemerkungen:

Diese Funktion stellt das Komplement von **TAN** dar. Sie erwartet einen "Tangens-Wert" als Argument und uebergibt einen "Winkel-Wert", naemlich den Winkel, der dem als Argument angegebenen Tangens entspricht. Das Ergebnis von **ATN** ist ein Wert doppelter Genauigkeit.

ATN arbeitet wie alle trigonometrischen Funktionen von TBASIC nicht mit der Einheit Grad (0...360), sondern mit **Rad** (0 bis 2 Pi), wobei der positive Teil der X-Achse dem Wert Ø entspricht. Gezaehlt wird in derselben Richtung wie bei Grad, naemlich entgegen dem Uhrzeigersinn: die positive Y-Achse ("nach oben", 90 Grad) entspricht dem Wert 1/2 Pi, die negative X-Achse ("nach links", 180 Grad) entspricht dem Wert Pi, die negative Y-Achse ("nach unten", 270 Grad) dem Wert 3*1/2 Pi. Ein Grad entspricht also Pi/180 Rad.



Kartesisches Koordinatensystem mit Grad/Rad

Wenn das Programm die Einheit Grad benoetigt, laesst sich eine einfache Umwandlung vornehmen:
Rad * 57.2958 ergibt Grad; Grad / 57.2958 ergibt Rad. Die Umrechnung eines Ergebnisses von **ATN** in Grad ergibt sich also zu

```
GradWert = ATN(TangensWert) * 57.2958
```

Der Wert von Pi betraegt bei einer Genauigkeit von 16 Dezimalstellen 3.141592653580793 und kann mit **ATN** ueber die folgende Formel berechnet werden:

```
Pi# = 4 * ATN(1)
```

Beispiel:

Ermittlung des Winkelwertes, dessen Tangens sich zu 0.24567 Rad ergibt:

```
Winkel = ATN(0.24567)
```

4.3.1.5. Zeichenkettenbezogene numerische Funktionen

- ASC

Uebergibt den ASCII-Code fuer das erste Zeichen der Zeichenkette x$.

Syntax: `v = ASC(x$)`

Bemerkungen:

ASC erwartet ein Argument vom Typ Zeichenkette und uebergibt den ASCII-Code im Bereich von 0...255.

- INSTR

Durchsucht eine Zeichenkette nach einer Zeichenfolge und uebergibt ihre Position innerhalb der Zeichenkette.

Syntax: `v = INSTR([n,]Zielzeichenkette,Suchzeichenkette)`

Bemerkungen:

n — ist ein Integerausdruck im Bereich 1...32767, der wahlweise angegeben werden kann und die Position bezeichnet, wo die Suche beginnen soll. Wenn n nicht angegeben ist, beginnt die Suche mit dem ersten Zeichen. **Zielzeichenkette** und **Suchzeichenkette** koennen Zeichenkettenvariablen, Zeichenketten-Ausdruecke oder Zeichenketten-Konstanten sein.
INSTR uebergibt die Position des ersten Zeichens der **Suchzeichenkette** innerhalb der **Zielzeichenkette**. Wenn die gesuchte Zeichenfolge innerhalb der Zielzeichenkette nicht gefunden wird, liefert **INSTR** den Wert Ø.

Wenn die **Suchzeichenkette** die Laenge Ø hat, uebergibt **INSTR** grundsaetzlich den Wert 1 - egal, wie lang die **Zielzeichenkette** ist.

Hinweis:

Bei der Suche bzw. dem Vergleich wird zwischen Gross- und Kleinbuchstaben unterschieden.

Unterschied:
Bei BASI kann n nur maximal 255 Zeichen sein.

- LEN

Uebergibt die Laenge der Zeichenkette x$.

Syntax: v = LEN(x$)

Bemerkungen:

Es wird ein Integerwert im Bereich 0...32767 uebergeben, der die Anzahl der Zeichen in x$ angibt.

Unterschied:
Bei BASI betraegt die maximale Laenge einer Zeichenkette 255 Zeichen.

- VAL

Uebergibt den numerischen Wert der Zeichenkette x$.

Syntax: v = VAL(x$)

Bemerkungen:

VAL analysiert eine Folge von numerischen ASCII-Zeichen (d.h. die Ziffern Ø bis 9, die Zeichen + und - sowie den Dezimalpunkt und die Buchstaben E und D) und berechnet daraus einen Wert.
Wenn x$ mit einem oder mehreren numerischen Zeichen beginnt, dann analysiert VAL bis zum ersten nicht-numerischen Zeichen und uebergibt den entsprechenden Wert.

Beginnt x$ dagegen nicht mit einem numerischen Zeichen, dann liefert VAL den Wert Ø zurueck - eine Fehlermeldung wird nicht angezeigt.

4.3.2. Zeichenkettenfunktionen

4.3.2.1. Allgemeine Zeichenkettenfunktionen

- CHR$

Uebergibt ein Zeichen, das dem im Argument angegebenen ASCII-Code entspricht.

Syntax: v$ = CHR$(n)

Bemerkungen:
keine

- **HEX$**

Uebergibt eine Zeichenkette, die den hexadezimalen Wert eines dezimalen Arguments darstellt.

Syntax: **v$ = HEX$(n)**

Bemerkungen:
keine

- **OCT$**

Uebergibt eine Zeichenkette, der den oktalen Wert eines dezimalen Arguments darstellt.

Syntax: **v$ = OCT$(n)**

Bemerkungen:
keine

- **BIN$**

Uebergibt den Wert des numerischen Arguments als eine "binaere Zeichenkette", der aus den Zeichen "0" und "1" besteht.

Syntax: **v$ = BIN$(n)**

Bemerkungen:

Das Argument wird durch Runden und Abschneiden der Nachkommastellen auf das Format Integer gebracht, es muss im Bereich von -32768 bis +65535 liegen. **BIN$** erzeugt dann eine Zeichenfolge, die der binaeren Darstellung des Arguments entspricht und liefert sie als Zeichenkette zurueck. Negative Argumente werden in Form ihres Zweierkomplements zurueckgeliefert.

Die Laenge der zurueckgelieferten Zeichenkette haengt von der Groesse des numerischen Arguments ab. **BIN$** liefert minimal 1, maximal 16 Zeichen.

Beispiel:

Diese Schleife gibt die binaeren Aequivalente der Zahlen von -6 bis +6 aus.

```
FOR I% = -6 TO 6
  PRINT "Der Wert von ";I%;"in binaerer ";
  PRINT "Form ist "; BIN$(I%)
NEXT I%
```

- STR$

Wandelt den Wert eines numerischen Ausdrucks in eine Zeichenkette um.

Syntax: **v$ = STR$(x)**

Bemerkungen:
keine

- LEFT$

Uebergibt den linken Teil einer Zeichenkette mit der Laenge n.

Syntax: **v$ = LEFT$(x$, n)**

Bemerkungen:

n ist ein Integerausdruck im Bereich 0...32767 und gibt an, wieviele Zeichen der Zeichenkette **x$** von links beginnend in **v$** eingetragen werden.

- RIGHT$

Uebergibt den rechten Teil einer Zeichenkette mit der Laenge n.

Syntax: **v$ = RIGHT$(x$, n)**

Bemerkungen:

n ist ein Integerausdruck im Bereich 0...32767 und legt fest, wieviel Zeichen der Zeichenkette **x$** von rechts in **v$** eingetragen werden sollen.

- MID$ (Funktion und Anweisung)

Uebergibt eine Anzahl von Zeichen aus der Mitte einer Zeichenkette.

Syntax: **Als Funktion: v$ = MID$(x$, n[,m])**
Als Anweisung: MID$(v$, n[,m]) = y$

Bemerkungen:

n ist ein Integer-Ausdruck im Bereich 1...32767.
m ist ein Integer-Ausdruck im Bereich 0...32767.

- SPACE$

Uebergibt eine Zeichenkette, die aus n Leerzeichen besteht.

Syntax: `v$ = SPACE$(n)`

Bemerkungen:

n ist ein Integerausdruck im Bereich 0...32767.

- STRING$

Uebergibt eine Zeichenkette der Laenge n, deren Zeichen alle aus dem ASCII-Code m oder dem ersten Zeichen von x$ bestehen.

Syntax:
```
v$ = STRING$(n, m) oder
v$ = STRING$(n, x$)
```

Bemerkungen:

n ist ein Integerausdruck im Bereich 1...32767.
m liegt im Bereich von 0...255.

4.3.2.2. Zeichenkettenfunktionen fuer Ein-/Ausgabe

- INSTAT

Uebergibt den momentanen Status der Tastatur bzw. des Tasaturpuffers.

Syntax: `v = INSTAT`

Bemerkungen:

INSTAT prueft, ob der Tastaturpuffer Zeichen enthaelt, und liefert einen Wahrheitswert zurueck:

```
-1  (TRUE)   bedeutet, dass seit dem letzten Lesevorgang
             mindestens eine Taste gedrueckt wurde.
 Ø  (FALSE)  signalisiert einen leeren Tastaturpuffer.
```

Ein Lesevorgang im eigentlichen Sinn findet dabei nicht statt. Um den Tastaturpuffer zu entleeren, kann z.B. INKEY$ verwendet werden.

Beispiel:

sh. INKEY$

- INKEY$

Lesen der Tastatur ohne gleichzeitige Anzeige der gelesenen Zeichen auf dem Bildschirm (Echo).

Syntax: **v$ = INKEY$**

Bemerkungen:

INKEY$ liefert saemtliche Tastendruecke in unveraenderter und uninterpretierter Form zurueck. Das gilt ebenso fuer **Control-Tasten** sowie **TAB**, **RETURN** und **BACKSPACE**. Drei Tastenkombinationen koennen allerdings auch ueber diese Funktion nicht erfasst werden:

<CTRL-BREAK> bricht die Programmausfuehrung ab, wenn beim Compilieren die Option **Keyboard break** auf **ON** gesetzt ist, ansonsten wird diese Tastenkombination ignoriert.

<CTRL-ALT-DEL> setzt das System zurueck.

<CTRL-PRTSC> druckt den Inhalt des Bildschirms aus.

Beispiel:

Das folgende Programm demonstriert zwei Anwendungen von **INKEY$**: In der ersten Schleife wird der Tastaturpuffer entleert - Zeichen, die der Benutzer bereits vor der Anforderung zur Eingabe eingegeben hat, werden ignoriert. Im 2. Teil wiederholt das Programm eingegebene Zeichen in endloser Folge auf dem Bildschirm - solange, bis die Taste <ESC> gedrueckt wird.

```
'Entleerung des Tastaturpuffers
WHILE INSTAT      'solange Zeichen anliegen,
   X$ = INKEY$    'ignorieren
WEND

'Aufforderung und Einlesen eines Zeichens
PRINT "Bitte geben Sie ein Zeichen ein ";
WHILE NOT INSTAT: WEND      'warten darauf
X$ = INKEY$                 'Zeichen lesen

DO
  PRINT X$;                 'immer wieder neu
  IF INSTAT THEN X$ = INKEY$
LOOP UNTIL X$ = CHR$(27)    'bis ESC gedrueckt'
                            'wird
END
```

- INPUT$

Liest eine bestimmte Anzahl Zeichen von der Tastatur oder aus einer Datei.

Syntax: **v$ = INPUT$(n[,[#]Dateinummer])**

Bemerkungen:

n liegt im Bereich 0...32767.
Hat **n** den Wert 0, uebergibt **INPUT$** eine Null-Zeichenkette.

Grenzen:
Die Funktionstasten und die Tasten zur Steuerung des Kursors liefern keine direkten ASCII-Codes. **INPUT$** speichert hier jeweils den Wert **CHR$(Ø)**. **INKEY$** hat diese Begrenzung nicht, dafuer muss aber fuer eine Eingabe mehrerer Zeichen eine entsprechende Schleife programmiert werden (sh. INKEY$).

Mit **INPUT$** kann eine sequentielle Datei bekannter Zeilenlaenge sehr einfach gelesen werden. Falls die Datei sehr grosse Datenbloecke enthaelt, die nicht nur aus ASCII-Zeichen bestehen, sollte die Datei im Modus **BINARY** eroeffnet und **GET$** benutzt werden.

4.3.3. Dienstleistungsfunktionen

- TIMER

Ermittelt die Anzahl der seit 0 Uhr ØØ bzw. seit dem Start des Computers vergangenen Sekunden.

Syntax: **v = TIMER**

Bemerkungen:

Der uebergebene Wert ist entweder

- die Uhrzeit in Sekunden (d.h. die Anzahl der seit 0 Uhr ØØ vergangenen Sekunden);

- oder die Anzahl von Sekunden, die seit dem Start des Systems vergangen sind. Das ist der Fall, wenn beim Systemstart keine Uhrzeit gesetzt wurde.

Die Nachkommastellen des von **TIMER** uebergebenen Wertes stehen fuer Zehntel- bzw. Hundertstelsekunden. Die Genauigkeit des uebergebenen Wertes liegt allerdings nur bei rund einer Zehntelsekunde.
Fuer genauere Zeitmessungen sollte **MTIMER** verwendet werden.

Beispiel:

```
PRINT "Haben Sie die Uhrzeit beim Start des ";
INPUT "Computers gesetzt (J/N) "; X$

IF UCASE$(X$) = "J" THEN
   PRINT "Wir haben jetzt "; TIMER;
   PRINT "Sekunden nach Mitternacht."
ELSE
   PRINT "Seit dem Start des Computers ";
   PRINT "sind "; TIMER; "Sekunden vergangen."
END IF
END
```

- LCASE$

Verwandelt Grossbuchstaben der Zeichenkette in Kleinbuchstaben.

Syntax: **x$ = LCASE$(y$)**

Bemerkungen:

In der von **LCASE$** erzeugten Zeichenkette sind saemtliche Grossbuchstaben des Arguments in Kleinbuchstaben umgewandelt worden, alle anderen Zeichen bleiben unveraendert. Die Zeichenkette kann eine maximale Laenge von 32767 Zeichen haben.

- UCASE$

Verwandelt Kleinbuchstaben der Zeichenkette in Grossbuchstaben.

Syntax: **x$ = UCASE$(y$)**

Bemerkungen:

UCASE$** wandelt saemtliche Kleinbuchstaben der Zeichenkette **y$ in Grossbuchstaben um. Alle anderen Zeichen bleiben unveraendert. **y$** kann eine Laenge zwischen 0 und 32767 Zeichen haben.

- RND

Uebergibt eine Zufallszahl zwischen 0 und 1.

Syntax: `v = RND[(x)]`

Bemerkungen:

Der durch **RND** uebergebene Wert ist eine Zahl doppelter Genauigkeit im Bereich von 0 bis 0.99999... - der Wert 1 wird von **RND** nie erreicht.
Um eine Folge von Integerwerten im Bereich von 1...n zu erhalten, kann folgende Formel benutzt werden:

```
ZUFALL% = INT(RND*n) +1
```

- FRE

Es wird der freie Speicherplatz ermittelt.

Syntax: `v = FRE(Zeichenketten-Ausdruck|-1|-2)`

Bemerkungen:

FRE ermittelt die Anzahl freier Bytes eines Speicherbereichs als numerischen Wert im Format Langinteger. Fuer Zeichenketten-Ausdruecke als Argument wird der freie Platz im Zeichenkettenspeicherbereich, fuer den Wert -1 der freie Platz im Feld-Speicherbereich und fuer den Wert -2 der freie Platz auf dem Stack ermittelt.
Zeichenketten-Ausdruck kann der Name einer beliebigen Zeichenkettenvariablen, eine Zeichenkettenfunktion oder eine Zeichenkettenkonstante sein. Ihr tatsaechlicher Wert ist bedeutungslos - hier geht es nur um den Typ des Arguments.

Grenzen:
Die Anzeige des freien Speicherbereichs auf dem Stack ist von der mit dem Computer-Befehl **$STACK** gesetzten Groesse abhaengig - ausgegeben wird die Byte-Zahl, die innerhalb des fuer den Stack reservierten Speicherbereichs noch verbleibt.

Unterschiede:
Die **FRE**-Funktion von TBASIC liefert erheblich mehr Informationen als die von BASI.
Bei BASI benoetigt die Funktion **FRE** rein aus syntaktischen Gruenden ein Argument. Eine Auswertung nach Typ oder Wert findet nicht statt. Ob **FRE(0)** oder **FRE(x$)** - zurueckgeliefert wird grundsaetzlich der freie Speicherplatz innerhalb des Datensegments. Da hier sowohl Felder als auch Zeichenketten gespeichert werden und rekursive Definitionen bei BASI nicht moeglich sind, ist auch weder eine entsprechende Unterteilung noch eine Pruefung des Stacks definiert.

- LBOUND

Uebergibt die kleinstmoegliche Indexnummer eines Feldes bzw. einer Felddimension.

Sytax: **LBOUND (Feldname[,Dimension])**

Bemerkungen:

Feldname ist eine Konstante und steht fuer den Namen des Feldes, der (optionale) numerische Ausdruck **Dimension** fuer die Dimension, deren kleinste Indexnummer ermittelt werden soll. Zulaessige Werte fuer **Dimension** liegen im Bereich 1 bis 8. Wenn dieser Parameter nicht angegeben wird, setzt TBASIC den Wert 1 ein.
Solange die **DIM**-Anweisung nicht zusammen mit einer Bereichsangabe (z.B. **DIM X%(50:60)**) benutzt wurde oder ein Minimalindex mit **OPTION BASE** gesetzt ist, gibt TBASIC dem ersten Element eines Feldes die Indexnummer Ø.

Die Funktion **UBOUND** liefert die Indexnummer des hoechsten Elements einer Feld-Dimension.

Beispiel:

```
DIM X1%(20)              'Untergrenze ist Ø
DIM X2%(10:30)           'Untergrenze ist 10
DIM X3%(10,12:14)        'Untergrenze ist 0 und 12

PRINT LBOUND (X1%)       'ergibt Ø
PRINT LBOUND (X2%)       'ergibt 10
PRINT LBOUND (X3%,1)     'ergibt Ø (1. Dimension)
PRINT LBOUND (X3%,2)     'ergibt 12 (2. Dimension)
```

- UBOUND

Ermittelt die Nummer des hoechsten Elements eines Feldes bzw. einer Felddimension.

Syntax: **v = UBOUND(Feldname[,Dimension])**

Bemerkungen:

Feldname ist eine Konstante und steht fuer den Namen des Feldes. Der (optionale) Zusatz **Dimension** ist ein numerischer Ausdruck und gibt die Nummer der Dimension an, deren hoechste Indexnummer **UBOUND** ermitteln soll. Zulaessige Werte fuer **Dimension** liegen im Bereich von 1 bis 8; wenn diese Angabe fehlt, setzt TBASIC den Wert 1 ein.

Zusammen mit der Funktion **LBOUND**, die die niedrigste Indexnummer eines Feldes ermittelt, kann die Elementenzahl eines Feldes sozusagen nachtraeglich ermittelt werden.

Beispiel:

```
DIM A%(10)            'Indexnummern von 0 bis 10
DIM B(7:12)           'Indexnummern von 7 bis 12
OPTION BASE 1
DIM C$(8,10:17)       'zweidimensional

PRINT UBOUND (A%)       'ergibt 10
PRINT UBOUND (B)        'ergibt 12
PRINT UBOUND (C$,1)     'ergibt 8
PRINT "Die 2. Dimension des Feldes C$ hat"; _
      UBOUND(C$,2) - LBOUND(C$,2) "Elemente."
```

4.3.4. Funktionen zur Drucker- und Bildschirmsteuerung

- TAB

Setzt den Kursor auf die angegebene Position n.

Syntax: **PRINT TAB(n)**

Bemerkungen:
keine

- SPC

Uebergibt n Leerzeichen.

Syntax: **PRINT SPC(n)**

Bemerkungen:
keine

- POS

Ermittelt die momentane Spaltenposition des Kursors.

Syntax: **v = POS(n)**

Bemerkungen:
siehe Pkt. 4.6.1.

- LPOS

Ermittelt die momentane Druckposition im Druckpuffer.

Syntax: **v = LPOS(n)**

Bemerkungen:
keine

4.4. Dateiarbeit

4.4.1. Namensgebung fuer Dateien

Eine Dateiangabe muss den Regeln von DCP entsprechen und besteht aus folgenden Angaben:

[Pfad] Dateiangabe

Mit der wahlfreien Angabe von Pfad wird das entsprechende Verzeichnis ausgewaehlt. Dateiangabe waehlt die Datei aus und besteht aus den Angaben

Dateiname.Dateityp

Wobei Dateiname eine Folge von max. 8 Zeichen ist. Dateityp ist eine (optionale) Typ-Kennzeichnung, die max. aus 3 Zeichen bestehen darf.

Dateiname und Dateityp duerfen folgende Zeichen enthalten:

A-Z 0-9 () {} @ # $ % & / ! - _ '

Leerzeichen ist nicht zugelassen, Kleinbuchstaben werden automatisch in Grossbuchstaben umgewandelt.

Im Gegensatz zu Marken muss das erste Zeichen nicht unbedingt ein Buchstabe sein.

Weitere Angaben zu Dateiangaben und Pfadnamen siehe Dokumentation "Bedienungsanleitung und Sprachbeschreibung fuer BASIC-Interpreter (BASI)" Kapitel 6.

4.4.2. Eroeffnen und Schliessen von Diskettendateien

- OPEN

Eroeffnen einer Datei fuer nachfolgende Lese- und Schreibaktionen.

Syntax: **OPEN Dateiangabe [FOR Modus] AS [#]Dateinummer**
[LEN = Satzlaenge]

oder

OPEN Modus2, [#]Dateinummer, Dateiangabe
[,Satzlaenge]

Bemerkungen:

Modus: Zeichenkettenausdruck, der folgende Werte annehmen kann:

OUTPUT	Sequentielle Ausgabe. Der alte Inhalt der Datei wird geloescht.
INPUT	Sequentielle Eingabe
APPEND	Sequentielle Ausgabe. Daten werden an den alten Inhalt angehaengt.
RANDOM	Direktzugriffsdateien. Lesen und Schreiben erlaubt.
BINARY	Binaerdatei, unstrukturiert. Lesen und Schreiben erlaubt.

Modus2: Zeichenkettenausdruck, dessen erstes Zeichen folgende Werte annehmen kann:

0	**OUTPUT**
I	**INPUT**
A	**APPEND**
R	**RANDOM**
B	**BINARY**

Die Angabe LEN = Satzlaenge sollte bei Direktzugriffsdateien immer vorhanden sein.

- CLOSE

Schliessen der Ein-/Ausgabe zu einer Datei.

Syntax: **CLOSE [[#]Dateinummer[,[#]Dateinummer]...]**

Bemerkungen:
keine

4.4.3. Sequentielle Dateien

- PRINT # und PRINT # USING

Schreiben von Daten in eine sequentielle Datei.

Syntax: **PRINT #Dateinummer, [USING V$;]**
Liste von Ausdruecken [;]

Bemerkungen:

Beim Aufzeichnen von einzelnen Datenfeldern ist es notwendig, Kommas mit aufzuzeichnen, wenn diese Datenfelder spaeter mit der Anweisung INPUT # gelesen werden sollen, da die Anweisung INPUT # ein Komma als Trennzeichen zwischen den einzelnen Datenfeldern (Datenworten) erwartet.

Beispiel:

```
PRINT #1, 1 "," 2 "," 3
```

Diese Zeile laesst sich mit einer entsprechenden INPUT # - Anweisung (z.B. INPUT #1,a,b,c) wieder direkt zurueecklesen.

Wesentlich einfacher geht das Schreiben mit der Anweisung WRITE #. Hier werden die Trennzeichen automatisch von TBASIC gesetzt.

- WRITE #

Schreiben von Daten in eine sequentielle Datei.

Syntax: **WRITE #Dateinummer, Liste der Ausdruecke**

Bemerkungen:
keine

- INPUT #

Einlesen von Daten aus einer sequentiellen Datei.

Syntax: **INPUT #Dateinummer, Variablenliste**

Bemerkungen:
keine

- LINE INPUT #

Einlesen einer Zeile von einer sequentiellen Datei in eine Zeichenkettenvariable. Trennzeichen werden ignoriert.

Syntax: **LINE INPUT #Dateinummer, Zeichenkettenvariable**

Bemerkungen:
keine

4.4.4. Direktzugriffsdateien

- FIELD

Definition von Feldvariablen als Puffer einer Direktzugriffsdatei.

Syntax: **FIELD [#]Dateinummer, Laenge AS Zeichenkettenvariable**
[,Laenge AS Zeichenkettenvariable] ...

Bemerkungen:
keine

- LSET und RSET

Uebergeben von Daten in den Puffer einer Direktzugriffsdatei.

Syntax: **LSET Zeichenkettenvariable = Zeichenkettenausdruck**
RSET Zeichenkettenvariable = Zeichenkettenausdruck

Bemerkungen:
keine

- PUT

Schreiben eines Datensatzes aus dem Puffer fuer Direktzugriff in eine Direktzugriffsdatei.

Syntax: **PUT [#]Dateinummer [,Satznummer]**

Bemerkungen:
keine

- GET

Lesen eines Datensatzes aus einer Direktzugriffsdatei in einen Puffer fuer Direktzugriff.

Syntax: **GET [#]Dateinummer [,Satznummer]**

Bemerkungen:

Das Lesen eines nichtexistierenden Datensatzes mit GET erzeugt keine Fehlermeldung. TBASIC setzt in diesem Fall fuer numerische Variablen den Wert 0, fuer Zeichenkettenvariablen eine Nullzeichenkette ein.

4.4.5. Binaerdateien

- SEEK

Setzen der Position innerhalb einer als BINARY eroeffneten Datei fuer ein nachfolgendes Lesen oder Schreiben.

Syntax: **SEEK [#]Dateinummer, Position**

Bemerkungen:

Dateinummer ist ein Integer-Ausdruck und stellt die Nummer dar, die der Datei bei ihrer Eroeffnung zugeordnet wurde. Die Datei muss im Modus BINARY eroeffnet worden sein.

Position ist ein ganzzahliger Ausdruck im Bereich von 1 bis 16777215 und setzt die Position innerhalb der Datei entsprechend: Wird hier der Wert 1 angegeben, liest ein folgendes GET$ das erste Byte der Datei bzw. ein folgendes PUT$ ueberschreibt das erste Byte der Datei; fuer **Position** = 33 wird ab dem 33. Byte der Datei gelesen/geschrieben usw. Die Position innerhalb der Datei wird durch GET$ bzw. PUT$ um die Anzahl der gelesenen/geschriebenen Bytes erhoeht.

Beispiel:

Die erste Anweisung GET$ liest das 10. und 11. Byte der Datei BINTEST. Die zweite Anweisung GET$ liest die Bytes 12, 13 und 14. Anschliessend steht die Position auf dem 15. Byte.

```
OPEN "BINTEST." FOR BINARY AS #1
SEEK 1, 10
GET$ 1, 2, A$          'liest zwei Zeichen
GET$ 1, 3, B$          'drei Zeichen
```

Direkt nach der Eroeffnung einer Datei als BINARY steht die Position auf dem ersten Byte, mit der Funktion LOC kann ihr momentaner Wert zu jedem Zeitpunkt gelesen werden.

Hinweis:

Mit SEEK kann jederzeit ueber das momentane Dateiende hinaus positioniert werden. Eine Befehlsfolge wie

```
OPEN "BIN" FOR BINARY AS #1          'neue Datei
SEEK 1, 500
```

setzt tatsaechlich die "Position" 500 innerhalb der Datei. LOC(1) liefert auch diesen Wert zurueck, EOF(1) liefert TRUE. Eine folgende GET$-Anweisung liefert eine Nullzeichenkette (keine Fehlermeldung!). Wird die Datei danach wieder geschlossen, zeigt der Directory-Eintrag eine Laenge von 0. Erst eine tatsaechliche Schreibaktion wie PUT$ 1, "XXX" belegt die entsprechende Anzahl von Bytes auf der Diskette.

- PUT$

Schreiben von Daten in eine Binaerdatei.

Syntax: **PUT$ [#]Dateinummer, Zeichenkettenausdruck**

Bemerkungen:

Dateinummer ist ein Integer-Ausdruck und stellt die Nummer dar, die der Datei bei ihrer Eroeffnung zugeordnet wurde. Die Datei muss im Modus BINARY eroeffnet worden sein.

Zeichenketten-Ausdruck
kann eine Konstante, ein Ausdruck oder eine Funktion sein und enthaelt die Daten, die ab der momentanen Position innerhalb der Datei geschrieben werden sollen.

Ueber die Anweisung SEEK kann diese Position gesetzt werden, sie wird jeweils um die Anzahl der mit PUT$ geschriebenen Bytes erhoeht. Wurde mit SEEK zuvor ueber das Dateiende hinaus positioniert, wird die Datei entsprechend vergroessert.

- GET$

Lesen einer Zeichenkette von einer Binaerdatei.

Syntax: **GET$ [#]Dateinummer, Anzahl, Zeichenkettenvariable**

Bemerkungen:

Dateinummer ist ein Integer-Ausdruck und stellt die Nummer dar, die der Datei bei ihrer Eroeffnung zugeordnet wurde.

Anzahl ist ein Integer-Ausdruck im Bereich vom 0 bis 32767 und gibt die Anzahl der zu lesenden Bytes an.

Zeichenkettenvariable
Ist der Name einer ZK-Variablen, in der die gelesenen Bytes gespeichert werden sollen.

GET$ liest ab der momentanen Position in der Datei (die zuvor mit SEEK gesetzt werden kann). Die Position in der Datei wird automatisch um die Anzahl der gelesenen Bytes erhoeht.

Grenzen:
Die Anzahl gelesener Bytes wird automatisch durch das physikalische Ende der Datei begrenzt. Befindet sich die Position in der Datei beispielsweise 320 Byte vor dem Dateiende, dann liefert eine Anweisung wie

```
GET$ #1, 500, Puffer$
```

nur 320 Byte zurueck. Befindet sich die Position in der Datei bereits vor einem Leseversuch auf dem Dateiende, dann liefert ein erneutes Lesen mit GET$ eine Nullzeichenkette zurueck (keine Fehlermeldung!).

Beispiel:

```
SUB Schreiben
LOCAL i%
    OPEN "BIN" FOR BINARY AS #1
    FOR I% = 1  TO 255
        PUT$  #1, CHR$(I%)
    NEXT I%
    CLOSE #1
END SUB

SUB Lesen  (Start%, Anzahl%)
LOCAL A$
    OPEN "BIN" FOR BINARY AS #1
    SEEK #1, Start%
    GET$ #1, Anzahl%, A$
    CLOSE #1
    PRINT  A$
END SUB


'Hauptprogramm

CALL Schreiben
Input "Position, Anzahl"; Pos%, Anz%
CALL Lesen (Pos%, Anz%)
END
```

4.4.6. Funktionen

- CVI, CVL, CVS, CVD

Umwandeln von Daten aus einer Direktzugriffsdatei in die numerische Form.

Syntax:

```
V% = CVI(2-Byte-Zeichenkette)
V& = CVL(4-Byte-Zeichenkette)
V! = CVS(4-Byte-Zeichenkette)
V# = CVD(8-Byte-Zeichenkette)
```

Bemerkungen:

CVI konvertiert eine Zeichenkette von 2 Byte Laenge in einen Integerwert; CVL konvertiert eine Zeichenkette von 4 Byte Laenge in einen Langintegerwert; CVS konvertiert eine Zeichenkette von 4 Byte Laenge in einen einfachgenauen Realwert; CVD konvertiert eine Zeichenkette von 8 Byte Laenge in einen doppeltgenauen Realwert.

TBASIC speichert in einer Direktzugriffs-Datei die einzelnen Bytes, aus denen sich der Wert einer numerischen Variablen ergibt, ohne diese Bytes in irgendeiner Form zu veraendern. Eine Konvertierung im eigentlichen Sinne findet durch diese vier Funktionen also nicht statt, es handelt sich in allen Faellen lediglich um ein direktes Lesen der entsprechenden Bytes, die bei einer Zuweisung in unveraenderter Form in den Speicherplatz der angegebenen Variablen kopiert werden.

Alle vier Funktionen duerfen nur im Zusammenhang mit dem Lesen von Direktzugriffs-Dateien verwendet werden!

- CVMD, CVMS

Konvertieren der Inhalte von Direktzugriffsdateien, die im Format des BASIC - Interpreters BASI aufgezeichnet wurden, in numerische Form.

Syntax: **y! = CVMS(4-Byte-Zeichenkette)**
y# = CVMD(8-Byte-Zeichenkette)

Bemerkungen:

CVMS wandelt eine Folge von vier Byte vom Format des BASIC-Interpreters BASI in das (IEEE-) Format einer einfachgenauen Realzahl von TBASIC um.

CVMD wandelt eine Folge von acht Byte vom Format des BASIC-Interpreters BASI in das (IEEE-) Format einer doppeltgenauen Realzahl von TBASIC um.

Beide Funktionen sind ausschliesslich fuer das Lesen von Direktzugriffs-Dateien vorgesehen, die mit einem interpretativen BASIC erzeugt wurden. Im Gegensatz zu den vorhergehenden Funktionen findet hier eine echte Umrechnung statt.

Beispiel:

Lesen einer Datei, die mit BASI erstellt wurde und schreiben in eine TBASIC-Datei. Voraussetzung dafuer ist, dass Struktur und Anzahl der Saetze in der Quelldatei bekannt sind.

```
OPEN "DATEI1.DAT" AS #1  LEN = 12        'BASI-Datei
OPEN "DATEI2.DAT" AS #2  LEN = 12        'TBASIC-Datei
FIELD #1, 4 AS BASIEREAL$, 8 AS BASIDREAL$
FIELD #2, 4 AS EREAL$, 8 AS DREAL$
FOR I% = 1 TO 10
     GET #1, I%
     X! = CVMS(BASIEREAL$)
     X# = CVMD(BASIDREAL$)
     PRINT X!, X#
     LSET EREAL$ = MKS$(X!)
     LSET DREAL$ = MKD$(X#)
     PUT #2, I%
NEXT I%
CLOSE
END
```

Die Umwandlung kann auch innerhalb eines Befehls vorgenommen werden:

```
LSET EREAL$ = MKS$(CVMS(BASIEREAL$))
LSET DREAL$ = MKD$(CVMD(BASIDREAL$)) .
```

- MKI$, MKL$, MKS$, MKD$

Umwandeln von numerischen Werten in Zeichenketten fuer das Schreiben in eine Direkzugriffsdatei.

Syntax:
```
V$ = MKI$(Integer-Ausdruck)
V$ = MKL$(Langinteger-Ausdruck)
V$ = MKS$(Real-Ausdruck einfacher Genauigkeit)
V$ = MKD$(Real-Ausdruck doppelter Genauigkeit)
```

Bemerkungen:

MKI$ liefert einen Integerwert als Zeichenkette von 2 Byte Laenge zurueck; MKL$ einen Langintegerwert als Zeichenkette mit 4 Byte; MKS$ einen einfachgenauen Realwert mit 4 Byte und MKD$ einen doppeltgenauen Realwert mit 8 Byte Zeichenkettenlaenge.

- MKMS$, MKMD$

Konvertieren von Realwerten in Zeichenketten fuer das Schreiben von Direkzugriffsdateien im Format, gefordert von BASI.

Syntax:
```
V$ = MKMS$(Realausdruck einfacher Genauigkeit)
V$ = MKMD$(Realausdruck doppelter Genauigkeit)
```

Bemerkungen:

MKMS$ rechnet einen einfachgenauen Realwert von TBASIC in das Format von BASI um und liefert eine Zeichenkette der Laenge 4 Byte.
MKMD$ rechnet einen doppeltgenauen Realwert um und liefert eine Zeichenkette der Laenge 8 Byte.

- LOC

Uebergibt die aktuelle Position in der Datei.

Syntax:
```
V = LOC(Dateinummer)
```

Bemerkungen:

Dateinummer Ist ein Integer-Ausdruck und steht fuer die Nummer, die der Datei bei ihrer Eroeffnung zugeordnet wurde.

LOC liefert einen ganzzahligen Wert zurueck, der abhaengig vom Typ der Datei (bzw. dem Modus der Eroeffnung) interpretiert werden muss:

- fuer Dateien, die mit BINARY eroeffnet wurden, wird in einzelnen Bytes gezaehlt. Der von LOC zurueckgelieferte Wert bedeutet, dass die momentane Position sich auf dem Byte innerhalb der Datei befindet. SEEK zaehlt in derselben Weise beim Setzen einer Position.
  LOC liefert die Anzahl der gelesenen Bytes zurueck, das bedeutet direkt nach dem Eroeffnen einer Binaerdatei den Wert 0.

- fuer sequentielle Dateien wird in "Saetzen" mit jeweils 128 Bytes gezaehlt.
  LOC liefert direkt nach der Eroeffnung der Datei den Wert 1 zurueck, nach dem Lesen oder Schreiben der ersten 128 Bytes den Wert 2 usw.

- fuer Direktzugriffs-Dateien wird die Nummer des zuletzt gelesenen oder geschriebenen Satzes zurueckgeliefert.

- fuer Kommunikationsdateien (COMn:) liefert LOC die Anzahl der noch nicht gelesenen Bytes im Eingabepuffer.

Beispiel:

```
OPEN "TEST.DAT"  FOR BINARY  AS #1
PRINT "Position = : " ;  LOC(1)
PUT$ #1, "Hallo"
PRINT "Position neu = : "; LOC(1)
CLOSE
END
```

- LOF

Uebergibt die Groesse einer Datei.

Syntax: **V = LOF(Dateinummer)**

Bemerkungen:
keine

- EOF

Zeigt an, ob das Ende der Datei erreicht wurde.

Syntax: **V = EOF(Dateinummer)**

Bemerkungen:
keine

4.5. Datenfernverarbeitung

Es ist moeglich, die Schnittstelle V.24 asynchrone Datenuebertragung ueber TBASIC zu unterstuetzen. Dazu ist in der Dokumentation BASIC-Interpreter BASI, Kapitel 7, nachzulesen.
Fuer die Ein- und Ausgabe der Daten sowie das Schliessen der Datei sind die Anweisungen und Funktionen der Dateiarbeit (Kapitel 4.4. dieser Sprachbeschreibung) zu nutzen.
Nur fuer das Eroeffnen einer Datei sowie fuer die Unterbrechungsabfrage existieren spezielle Anweisungen.

4.5.1. Eroeffnen einer Datenfernverarbeitungsdatei

- OPEN "COM

Eroeffnet eine Datei fuer Datenfernverarbeitung.

Syntax:

```
OPEN "COMn:[Geschwindigkeit][,Paritaet]
[,Daten][,Stop][,Optionen]"
AS [#]Dateinummer [LEN=Groesse]
```

Bemerkungen:

n ist die Nummer des seriellen Ports (1 oder 2)

Geschwindigkeit ist eine Konstante, mit der die Uebertragungsgeschwindigkeit (Baudrate) angegeben wird. Moegliche Werte sind 75, 110, 150, 300, 600, 1200, 2400, 4800 und 9600 bps. Die Standardvorgabe von TBASIC ist 300.

Paritaet legt das Pruefbit fuer die Uebertragung eines Datenwortes fest. Moegliche Angaben sind:

S	"Space"	(Paritaetsbit immer "0")
O	"Odd"	- ungerade Paritaet
M	"Mark"	(Paritaetsbit immer "1")
E	"Even"	- gerade Paritaet
N	"None"	- kein Pruefbit

Kein Pruefbit bedeutet hier: TBASIC sendet keine Paritaetsbits, eventuell empfangene Paritaetsbits werden ignoriert. Die Standardvorgabe ist "Even", d.h. gerade Paritaet.

Daten ist eine Konstante im Bereich von 5 bis 8 und gibt an, aus wievielen Bits ein Datenwort besteht. Die Standardvorgabe ist 7.

Stop ist eine Konstante und legt die Anzahl der Stop-Bits pro Datenwort fest. Moegliche Angaben sind 1 oder 2, die Standardvorgabe fuer die Baudraten 75 und 110 ist 2, fuer alle anderen Baudraten 1.

Dateinummer ist ein Integerausdruck und wird dem Port bei der Eroeffnung (in derselben Weise wie bei Diskettendateien) zugeordnet. Alle folgenden Operationen laufen ueber diese Nummer.

Groesse ist ein Integerausdruck und bezeichnet die maximale Anzahl von Bytes, die mit einem GET-Befehl gelesen bzw. mit einem PUT-Befehl geschrieben werden koennen. Die Standardvorgabe ist 128. Ein hier angegebener Wert darf die (zuvor mit dem Compiler-Befehl $COM festgelegte) Groesse des Port-Puffers nicht ueberschreiten.

Optionen **[,RS][,CS[msec]][,DS[msec]]**
[,CD[msec]][,LF][,PE]

Damit lassen sich weitere Einstellungen kontrollieren.

RS unterdrueckt Signale der Leitung RTS.
CS[msec] setzt die Verzoegerungszeit der Leitung CTS.
DS[msec] setzt die Verzoegerungszeit der Leitung DSR.
CD[msec] setzt die Verzoegerungszeit der Leitung CD.
Die Angabe von **LF** bewirkt, dass jedem gesendeten CR (Wagenruecklauf) automatisch ein LF (Zeilenvorschub) angefuegt wird. Die Angabe von **PE** aktiviert die Paritaetspruefung . Das jeweilige Argument **msec** legt die Zeit der Millisekunden fest, die TBASIC wartet, bevor der Fehler Timeout ("Auszeit") gemeldet wird. Moegliche Werte liegen im Bereich von 0 bis 65535. Wenn hier der Wert 0 oder gar nichts angegeben wird, fuehrt TBASIC keine Pruefung des entsprechenden Leitungszustandes aus. Die Standardvorgabe fuer **msec** ist 1000 fuer CS und DS, fuer CD ist sie 0.

Beispiel:

```
'Ein Zeichenkettenfeld zur Speicherung
'empfangener Zeichen:
DIM Eingabe$(1)

$COM1 1024 'fuer den Compiler: 1 KByte Puffer

'Eroeffnen von Port1 als COM-Datei
OPEN "COM1" AS #1 LEN =1

PRINT "Jeder Tastendruck beendet das " _
      "Programm..."

WHILE NOT INSTAT     'solange kein Tastendruck
      IF NOT LOF(1) THEN
         Eingabe$(0) = INPUT$(1,LOF(1))
         PRINT "Empfangene Zeichen: ";
         PRINT Eingabe$(0)
      END IF
WEND
END
```

4.5.2. Unterbrechungsabfrage

- ON COM(n)

Festlegen eines Unterprogramms zur Behandlung von Zeichen, die ueber die seriellen Ports empfangen werden.

Syntax: **ON COM(n) GOSUB Label**

Bemerkungen:

n — ist ein Integer-Ausdruck und steht fuer die Nummer des seriellen Ports (1 oder 2).

Label — bezeichnet das Unterprogramm, das aufgerufen werden soll, wenn ueber das entsprechende Port ein Zeichen empfangen wird.

Die Angabe der Zeilennummer 0 fuer **Label** hat eine spezielle Wirkung: sie macht eine vorherige ON COM-Anweisung rueckgaengig.

ON COM legt lediglich eine Zuordnung fest. Ein Aufruf des Unterprogramms durch empfangene Zeichen erfolgt erst dann, wenn die staendige Ueberpruefung des entsprechenden Ports mit dem Befehl **COM(n) ON** aktiviert worden ist.

Nach der Festlegung des Unterprogramms und dem Befehl COM(n) ON prueft TBASIC vor der Ausfuehrung jedes Programmabschnitts, ob ein Zeichen empfangen wurde. Ist das nicht der Fall, wird das Programm direkt fortgesetzt. Ansonsten wird ein Aufruf des Unterprogramms eingeschoben, bevor es mit dem naechsten Programmschritt weitergeht.

- COM(n)

Kontrolle der Reaktion des Programms auf Eingaben ueber die seriellen Ports.

Syntax: **COM(n) ON/OFF/STOP**

Bemerkungen:

n — Nummer des seriellen Ports (1 oder 2).

Mit diesem Befehl kann festgelegt werden, wie eine zuvor mit ON COM(n) GOSUB gesetzte "Abfangroutine" fuer Eingaben ueber die seriellen Ports genutzt wird.

- **ON** bewirkt, dass TBASIC nach jedem Ausfuehren eines Befehls prueft, ob ein Zeichen an dem entsprechenden Port anliegt. Wenn ja, wird die mit ON COM(n) GOSUB geschriebene Routine aufgerufen.

- **OFF** fuehrt dazu, dass saemtliche Aktivitaeten des betreffenden Ports ignoriert werden. Eventuell empfangene Zeichen gehen verloren.

- **STOP** bewirkt nur eine zeitweilige Unterbrechung. Die Routine wird zwar nicht aufgerufen, empfangene Zeichen werden aber gespeichert. Der naechste ON-Befehl erzeugt einen sofortigen Aufruf der Behandlungsroutine, wenn seit dem STOP-Befehl Zeichen empfangen wurden.

Beispiel:

Das folgende Programm bringt natuerlich nur dann sichtbare Ergebnisse, wenn der Computer ueber COM1: auch Zeichen empfaengt.

```
'Umleitung empfangener Zeichen auf die
'Routine COM1Empfang
ON COM(1) GOSUB COM1Empfang

'Bereitstellen eines Zeichenkettenfeldes zum
'Speichern empfangener Zeichen mit
'COM1Empfang:
DIM INPUFFER(5*1024)      '5 KByte

'Ein Zeiger auf den Beginn innerhalb dieses
'Puffers, ein weiterer auf sein momentanes Ende:
ZStart%=0:ZEnde%=0

'Compiler-Befehl: Puffer fuer COM1: 1KByte
$COM1 1024

'Zuordnen einer Dateivariablen zu COM1:
'und Eroeffnen als Datei:
OPEN "COM1:" AS #1

'und Aktivieren von COM1Empfang
COM(1) ON

'Hier folgt jetzt der Kern des Programms, der
'unabhaengig von COM1Empfang laeuft. Bei jedem
'Zeichenempfang wird COM1Empfang automatisch
'aufgerufen und danach die Programmausfuehrung
'so fortgesetzt, als waere nichts geschehen:
PRINT "Jeder Tastendruck beendet das Programm."
WHILE NOT INSTAT 'solange kein Tastendruck
      IF ZStart% <> ZEnde% THEN 'Empfang?
         PRINT "mit COM1: "; INPuffer(ZStart%)
         ZStart% = ZStart%+1
      END IF
      LOCATE 2,1
      PRINT TIME$
WEND
CLOSE             'COM1: schliessen
END               'Programmende bei Tastendruck
```

```
COM1Empfang:      'verarbeitet COM-Eingaben
    INPUT #1, INPuffer(ZEnde%)
    ZEnde% = ZEnde% + 1
RETURN
```

4.6. Grafik

Folgende Bildschirmadapter werden von TBASIC unterstuetzt:

Adapter	Modus	Bemerkungen
Monochrom	0	80x25 (40x25) Zeichen mit vier Attributen (normal, intensiv, blinkend, unterstrichen)
CGA	0, 1, 2	Textmodus: 80x25 (40x25) Zeichen bis zu 16 Farben Grafikmodus: 320x200 (640x200) Pixel bis zu vier Farben
EGA	0-2, 7-10	Textmodus: 80x25 (40x25) Zeichen Grafikmodus: bis zu 640x350 Pixel bis zu 16 Farben aus einer Palette von 64

Der auszuwaehlende Modus ist abhaengig von der im Computer installierten Grafikkarte.

4.6.1. Allgemeingueltige Anweisungen

- CLS

Loeschen des Bildschirms.

Syntax: **CLS**

Bemerkungen:

Im Grafikmodus wird der Bildschirm geloescht und der Kursor in die Mitte des Bildschirms gesetzt, je nach Aufloesung auf die Koordinaten (160, 100), (320, 100) oder (320, 175).

- SCREEN

Setzen der Bildschirmattribute.

Syntax: **SCREEN[Modus][,[Aendern][,[Aseite][,Vseite]]]**

Bemerkungen:

Modus	Ganzzahliger numerischer Ausdruck im Bereich 0...10. Gueltige Modi sind:
0	Textmodus mit der aktuellen Breite (40 oder 80)
1	Grafikmodus CGA, mittlere Aufloesung (320x200)
2	Grafikmodus CGA, hohe Aufloesung (640x200)
7	Grafikmodus EGA, mittlere Aufloesung (320x200), 16 Farben
8	Grafikmodus EGA, hohe Aufloesung (640x200), 16 Farben
9	EGA-Grafik mit EGA-Monitor: je nach Speicher 4 bis 16 aus 64 Farben, 640x350
10	EGA-Grafik mit monochromem EGA-Monitor: 4 Attribute, 640x350
Aseite	aktive Seite: Ist nur gueltig im Modus 0, 7, 8, 9 oder 10.
Vseite	visuelle Seite: Ist nur gueltig im Modus 0, 7, 8, 9 oder 10.

- LOCATE

Setzen des Kursors auf dem Bildschirm und Festlegen der Groesse des Kursors.

Syntax: **LOCATE [Zeile][,[Spalte][,[Anzeige][,[Start][,Stop]]]]**

Bemerkungen:
keine

- CSRLIN

Uebergeben der vertikalen Koordinate (Zeilenposition) des Kursors.

Syntax: **v = CSRLIN**

Bemerkungen:
keine

- POS

Uebergeben der Spaltenposition des Kursors.

Syntax: **v = POS(n)**

Bemerkungen:
keine

4.6.2. Farbe

4.6.2.1. Farbe im Textmodus - COLOR

Syntax: **COLOR [Vordergrund][,[Hintergrund]**
[,Bildschirmrand]]

Bemerkungen:

Bei Verwendung der EGA-Adapterkarte und des entsprechenden Monitors stehen 16 Farben aus einer Palette von 64 zur Verfuegung (siehe Anweisung Palette).
Der Parameter **Bildschirmrand** hat in diesem Fall keine Wirkung.

4.6.2.2. Farbe im Grafikmodus - COLOR

Syntax: nach SCREEN 1:
COLOR [Hintergrund][,[Palette]]

nach SCREEN 7...SCREEN 10:
COLOR [Vordergrund][,[Hintergrund]]

Bemerkungen:

Hintergrund Ganzzahliger numerischer Ausdruck im Bereich von 0 bis 63.

Vordergrund Ganzzahliger numerischer Ausdruck im Bereich von 1 bis 15.

SCREEN 1
Die tatsaechlich verwendete Hintergrundfarbe ergibt sich aus dem Wert von **Hintergrund** Modulo 16. Zulaessige Angaben fuer diesen Parameter liegen im Bereich von 0 bis 255. Die Werte 1, 16, 32 ... erzeugen jeweils die gleiche Hintergrundfarbe.

Die tatsaechlich verwendete Farb-Palette ergibt sich aus dem Wert von **Palette** Modulo 2. Zulaessige Angaben fuer diesen Parameter liegen im Bereich von 0 bis 255.

Es stehen folgende Zeichenfarben zur Verfuegung in Abhaengigkeit von der Farb-Palette:

Farbe	**Palette 0**	**Palette 1**
1	gruen	tuerkis
2	rot	violett
3	braun	weiss

Hinweis:

Um zwischen den beiden Paletten umschalten zu koennen, ist es in der TBASIC-Version 1.0 notwendig, nach Einstellen von SCREEN 1 eine Ausgabe auf das Port 3D8H zu bringen mit dem Wert 0AH. (OUT &H3D8H, &H0A).

Beispiel:

```
SCREEN 1
OUT &H3D8, &H0A
COLOR 0,0
CIRCLE (50,50),30,1
CIRCLE (100,100),30,2
CIRCLE (150,150),30,3
DELAY 2
COLOR 0,1
DELAY 2
```

SCREEN 7, SCREEN 8
Nur fuer EGA-Karten: Die Farben haengen von der momentan gesetzten Farb-Palette ab.
Die durch **Vordergrund** gesetzte Farbe wird sowohl fuer Zeichenanweisungen als auch zur Textdarstellung innerhalb des Grafikbildschirms verwendet. Zulaessige Werte gehen von 1 bis 15.
Fuer den **Hintergrund** stehen 16 Farben zur Verfuegung (Werte von 0 bis 15).

SCREEN 9
Die durch **Vordergrund** gesetzte Farbe wird sowohl fuer Zeichenanweisungen als auch zur Textdarstellung innerhalb des Grafikbildschirms verwendet. Zulaessige Werte gehen von 1 bis 15. Bei EGA-Karten, die lediglich ueber 64 KByte Bildspeicher verfuegen, koennen nur Werte von 1 bis 3 verwendet werden.
Fuer den **Hintergrund** stehen 64 Farben zur Verfuegung (Werte von 0 bis 63).

SCREEN 10
Zulaessige Werte fuer **Vordergrund** liegen im Bereich von 1 bis 3 und werden sowohl fuer Zeichenanweisungen als auch zur Textdarstellung verwendet. Der Wert 1 steht fuer schwarz, 2 fuer blinkend, 3 steht fuer hohe Intensitaet. Die Angabe des Wertes 0 erzeugt den Laufzeitfehler 5 (Illegaler Funktionsaufruf).

Mit **PALETTE** koennen die vorgegebenen Attribute geaendert werden. Die Zuordnungen sind:

0 aus
2 blinkend aus/ein
3 blinkend aus/intensiv
4 an
5 blinkend ein/intensiv
6 blinkend und intensiv/aus
7 blinkend und intensiv/ein
8 intensiv

Zulaessige Werte fuer den **Hintergrund** liegen im Bereich von 0 bis 8 und setzen eine dieser Hintergrundfarben.

Hinweis:

Die bei COLOR angegebene Hintergrundfarbe wird als Farbe 0 in die momentan gesetzte Palette eingetragen.
Wird bei einer Zeichenanweisung keine Farbe angegeben, benutzt TBASIC automatisch das maximal moegliche Attribut im gesetzten Modus.

4.6.3. Grafik

4.6.3.1. Anweisungen

- PSET und PRESET

Zeichnen eines Punktes auf dem Bildschirm.

Syntax: **PSET (x,y)[,Farbe]**
PRESET (x,y)[,Farbe]

Bemerkungen:

Farbe bestimmt das Attribut des zu zeichnenden Punktes. Welche Farbe durch ein bestimmtes Attribut erzeugt wird, haengt vom jeweiligen Grafikmodus und der momentan gesetzten Farb-Palette ab.

- LINE

Zeichnen einer Linie oder eines Vierecks auf dem Bildschirm.

Syntax: **LINE [(x1,y1)]-(x2,y2)[,[Farbe][,[B[F]][,Stil]]]**

Bemerkungen:

Farbe Ganzzahliger Ausdruck, legt die Farbe fest, in der gezeichnet werden soll.
Ist dieser Parameter nicht angegeben, benutzt TBASIC das maximal moegliche Attribut im gesetzten Grafikmodus.

- CIRCLE

Zeichnen einer Ellipse (Kreis) auf dem Bildschirm.

Syntax: **CIRCLE (x,y),r[,Farbe[,Start, End[,Bild]]]**

Bemerkungen:

Farbe Ganzzahliger numerischer Ausdruck, mit dem **Farbe** festgelegt werden kann.
Ist **Farbe** nicht angegeben, benutzt TBASIC den groessten zulaessigen Wert fuer den momentan gesetzten Modus.

Werte ausserhalb des zulaessigen Bereiches erzeugen den Laufzeitfehler 5 (Illegaler Funktionsaufruf).

- DRAW

Zeichnen eines grafischen Objektes auf dem Bildschirm.

Syntax: **DRAW Zeichenkette**

Bemerkungen:

Bei der Angabe von Variablennamen als Argumente der einzelnen Zeichenbefehle sollte die Form

```
= VARPTR$(Variable)
```

verwendet werden.

Beispiel:

```
DRAW "E=" + VARPTR$(K%)
```

- GET

Lesen von Punkten aus einem Bereich des Bildschirms in ein numerisches Feld.

Syntax: **GET (x1,y1) - (x2,y2), Bereich**

Bemerkungen:

Die Voraussetzung fuer eine Anwendung von GET ist ein numerisches Feld ausreichender Groesse, das sich mit der folgenden Formel berechnen laesst:

```
Bytezahl = 4 + INT (x * Bit-Anzahl + 7) / 8 * y
```

x steht dabei fuer die Pixelzahl in horizontaler Richtung (also die Differenz der Werte von x1 und x2), **y** fuer die Pixelzahl in vertikaler Richtung. **Bit-Anzahl** ist eine Konstante, deren Wert sich aus dem jeweiligen Grafikmodus ergibt:

SCREEN	Bit pro Pixel
1	2 (mittlere Aufloesung - 320x200)
2	1 (hohe Aufloesung - 640x200)
7	4 (EGA 320x200, 16 Farben)
8	4 (EGA 640x200, 16 Farben)
9	4 (EGA 640x350, 4...16 Farben)
10	2 (EGA Mono - 640x350)

- PUT

Faerben eines Bereiches des Bildschirmes.

Syntax: **PUT (x,y), Bereich[,Aktion]**

Bemerkungen:
keine

- PAINT

Fuellen eines Bildschirmbereiches mit Farbe.

Syntax: **PAINT (x,y)[,[Farbe],[Rand],[Hintergrund]**

Bemerkungen:

Farbe ist entweder ein numerischer Ausdruck, dann benutzt ihn TBASIC als Farb-Attribut (siehe COLOR, PALETTE) oder ein Zeichenketten-Ausdruck, der eine Bit-Folge festlegt, mit der die Flaeche ausgefuellt wird. Ist **Farbe** nicht angegeben, benutzt PAINT das hoechstmoegliche Attribut im momentan gesetzten Grafikmodus.

Rand bestimmt, durch welche bereits gezeichneten Linien die Ausfuehrung von PAINT begrenzt wird: Ueberall dort, wo TBASIC innerhalb des Bildspeichers der Grafik-Karte bereits gesetzte Punkte mit demselben Attribut wie dem Wert von **Rand** findet, wird das Ausfuellen in die betreffende Richtung beendet.

Hintergrund ist eine Zeichenkette mit einem Byte Laenge und legt eine weitere Maske fest, mit der durch **Rand** gesetzte Grenzen unter Umstaenden uebersprungen werden koennen.

Mit PAINT koennen umschlossene Flaechen gefuellt werden. Es spielt keine Rolle, wie komplex die Umschliessung aussehen mag. Es muss allerdings darauf geachtet werden, dass der zu fuellende Bereich wirklich komplett umschlossen ist.

Wenn der Parameter **Farbe** einen numerischen Wert hat, dann fuellt PAINT die Flaeche mit Pixeln dieses Attributs (die dabei entstehende Farbe haengt wie bei allen anderen Zeichenbefehlen auch vom momentan gesetzten Grafikmodus ab). Pixel mit dem durch **Rand** festgelegten Attribut und/oder Pixel desselben Wertes wie **Farbe** beenden den Vorgang.

Wird fuer **Farbe** dagegen ein Zeichenketten-Ausdruck angegeben, dann interpretiert PAINT diesen Wert als Folge einzelner Bits, die jeweils ein Byte (also 4 Pixel in der mittelhoch bzw. 8 Pixel in der hochaufgeloesten Grafik) breit sind; moegliche Laengen dieser Bitfolge liegen im Bereich von 1 bis 64 Byte. Das erste Byte von **Farbe** stellt also die erste Zeile des Fuellmusters dar, das zweite die zweite Zeile usw. Ein Muster in der hochaufloesenden Grafik wird beispielsweise folgendermassen definiert:

```
10101010    (erstes Byte:   Wert &HAA)
01010101    (zweites Byte:  Wert &H55)
```

Der benoetigte Wert fuer Farbe ist in diesem Fall eine Zeichenkette von zwei Byte Laenge. Die Anweisung muss demzufolge aussehen:

```
PAINT (50,50), CHR$(&HAA) + CHR$(&H55)
```

Hintergrund wird benutzt, um eine bereits mit einem Muster gefuellte Flaeche mit einem anderen Muster zu uebermalen. Wenn PAINT ein Pixel erreicht, dessen Position innerhalb eines Byte mit einem gesetzten Bit in **Hintergrund** zusammenfaellt, dann wird dieses Pixel nicht gegen **Rand** geprueft, PAINT arbeitet also in jedem Fall weiter.

Die Anzahl der moeglichen Farb-Attribute im jeweiligen Grafikmodus bestimmt letztendlich das optische Erscheinungs-bild einer Bitfolge, die fuer **Farbe** gesetzt wird - bei zwei moeglichen Attributen haben wir ein Bit pro Pixel, bei 16 moeglichen Attributen dagegen vier Bit. Mathematisch gesehen:

```
Bit-Pixel = LOG2 (moegliche Farb-Attribute)
```

PAINT nach SCREEN 1
In der mittelhoch aufloesenden Grafik stehen vier Farb-Attribute zur Verfuegung. Nach der zuvor angegebenen Formel hat jedes Pixel also 2 Bit. Je nachdem, ob die Farb-Palette 0 oder 1 gesetzt ist, kommen bei einer Fuellung mit PAINT die folgenden Farben heraus:

Farbe	**Attribut**	**"Farbe"**	**hexadezimal**
Palette 0			
gruen	01	01010101	&H55
rot	10	10101010	&HAA
braun	11	11111111	&HFF
Palette 1			
tuerkis	01	01010101	&H55
rosa	10	10101010	&HAA
weiss	11	11111111	&HFF

PAINT nach SCREEN 2
In der hochaufloesenden Grafik gibt es fuer jedes Pixel nur zwei moegliche Attribute, d.h. pro Pixel wird ein Bit zur Speicherung verwendet. Jedes Byte von **Farbe** legt also den Wert fuer acht aufeinanderfolgende Pixel fest: ein gesetztes Bit laesst einen Punkt erscheinen, ein geloeschtes Bit fuehrt zum Gegenteil. PAINT sorgt dafuer, dass angegebene Bitfolgen gleichmaessig innerhalb des umschlossenen Bereichs gezeichnet werden.

PAINT nach SCREEN 7,8,9 und 10
Hier ist die Erzeugung einer Farbe auf direktem Wege etwas komplizierter, weil EGA-Karten mehrere voneinander getrennte Speicherbereiche benutzen (drei Bereiche fuer die Komponente rot, gruen, blau und einen vierten fuer das Intensitaets-Bit). Um die Farbe von acht aufeinanderfolgenden Pixeln zu definieren, werden vier Byte benoetigt:

```
1010 1010          'erstes Byte  (Rot-Ebene)
1101 1010          'zweites Byte (Gruen-Ebene)
0110 1010          'drittes Byte (Blau-Ebene)
1101 1000          'viertes Byte (Intensitaets-Bit)
```

Das erste Pixel (ganz links aufgefuehrt) hat in diesem Beispiel einen Attribut-Wert von 11, das zweite einen von 14, das dritte 10 usw.

Im Modus 10 sind nur vier verschiedene Attribute moeglich, die durch zwei Speicherebenen dargestellt werden. Dementsprechend definieren hier zwei aufeinanderfolgende Bytes fuer **Farbe** die Attribut-Werte acht aufeinanderfolgender Pixel.

Beispiel:

```
SCREEN 1: CLS        'Grafik 320 * 200
'diagonale Linien:
LINIE$ = CHR$(&H40) + CHR$(&H40) + CHR$(&H10)_
             + CHR$(&H10) + CHR$(4) + CHR$(4)_
             + CHR$(1) + CHR$(1)
CIRCLE (100,100), 50
PAINT (100,100), LINIE$
```

- PALETTE / PALETTE USING

Zuordnung von Attribut-Werten und physikalischen Farben ueber eine Farb-Palette (nur EGA-Karten).

Syntax: **PALETTE (Attributwert, Farbe)**
PALETTE USING FELD(Index)

Bemerkungen:

EGA-Grafikkarten ordnen einem Pixel bestimmten Attributs nicht direkt eine Farbe zu, sondern schalten eine Farb-Palette dazwischen. Als Standardvorgabe ist der Eintrag 0 dieser Palette auf den Wert 0, der Eintrag 1 auf den Wert 1 usw. gesetzt (die Farbzuordnungen zu Pixel-Attributen bleiben also unveraendert).

Die Farbe bereits gezeichneter Pixel ist dadurch beliebig veraenderbar. Um beispielsweise saemtliche blauen Pixel (Attributwert 1) zeitweilig vom Bildschirm verschwinden zu lassen, reicht es, der Karte mitzuteilen, dass der Attributwert 1 die Farbe des Hintergrundes erzeugen soll, also:

```
PALETTE 1, 0      'alle Pixel mit Wert 1 werden schwarz
```

Attributwert ist ein **"Farbwert"**, d.h. ein Eintrag in der Farb-Palette, **Farbe** ist die tatsaechliche Farbe, die Pixeln dieses Attributwertes zugeordnet werden soll. (Welche physikalische Farbe erzeugt wird, haengt vom Bildschirm, der Grafik-Karte und dem gesetzten Modus ab.)

Wird der Befehl **PALETTE** ohne Argumente angegeben, stellt TBASIC die Standardzuordnungen wieder her (Attribut 0 --> 0, Attribut 1 --> 1 usw.).

Mit **PALETTE USING** lassen sich saemtliche Eintraege einer Farb-Palette ueber einen einzigen Befehl modifizieren. **Feld** steht fuer den Namen eines Integer-Feldes mit (mindestens) 16 Elementen, **Index** fuer einen Start-Index innerhalb dieses Feldes. Der Wert des Feld-Elements mit dem angegebenen **Index** wird von PALETTE USING als Farbe fuer das Pixel-Attribut 0 gesetzt, der Wert des Feld-Elements (Index + 1) als Farbe fuer das Pixel-Attribut 1 usw. Die Elemente des Feldes duerfen nur positive Werte enthalten - mit einer Ausnahme: Wenn ein Element den Wert -1 hat, bleibt der entsprechende Eintrag in der Farb-Palette unveraendert.

Beispiel:

```
DIM X%(15)       '16 Elemente
X%(0) = 8        'Attribut 0 --> Pixel-Wert 8
X%(1) = 0        'Attribut 1 --> Pixel-Wert 0
X%(2) = -1       'Attribut 2 bleibt unveraendert
X%(3) = 2        'Attribut 3 --> Pixel-Wert 2
.
.
.
PALETTE USING X%(0)     'setzt die Palette
```

Werte oberhalb von 15 lassen die dazugehoerigen Pixel vom Bildschirm verschwinden. Dadurch ist es beispielsweise moeglich, Texte unsichtbar vor einem Hintergrund ausgeben zu lassen und sie dann (durch eine Aenderung des Paletten-Eintrags) erscheinen zu lassen.

Moegliche Farben und Farbzuordnungen sind vom gesetzten Modus, der Ausfuehrung der EGA-Karte und dem benutzten Monitor abhaengig:

Farben und Attribut-Bereiche

Monitor	Modus	Attribut	Farbe
Mono/EGA	0	0 - 15	0 - 2
Color/EGA	0	0 - 31	0 - 15
Color/EGA	1	0 - 3	0 - 15
Color/EGA	2	0 - 1	0 - 15
Color/EGA	7	0 - 15	0 - 15
Color/EGA	8	0 - 15	0 - 15
EGA 64K	9	0 - 3	0 - 15
EGA >64K	9	0 - 15	0 - 63
Mono/EGA	10	0 - 3	0 - 8

Beispiel:

```
SCREEN 8              '640 * 200, 16 Farben
PALETTE               'Standardvorgabe

LINE (10,10) - (630,190),1,BF
DO
   FOR I% = 2 TO 14
       PALETTE 1,I%          'setzt Attribut 1 jeweils
                             'auf eine neue Farbe
       DELAY .5
   NEXT I%
LOOP UNTIL INSTAT            'bis zum naechsten
                             'Tastendruck
END
```

- VIEW

Definieren eines rechteckigen Bereiches auf dem Bildschirm.

Syntax: **VIEW [[SCREEN][(x1,y1) - (x2,y2)**
[,[Farbe][,Rand]]]]

Bemerkungen:

Ein mit VIEW definierter Bereich des Bildschirms hat die Eigenschaft, dass nur innerhalb dieser Flaeche gezeichnet werden kann. Ueber den Rand der Flaeche hinausgehende Linien und Objekte werden abgeschnitten.

- WINDOW

Neudefinieren der Bildschirmkoordinaten.

Syntax: **WINDOW [[SCREEN](x1,y1) - (x2,y2)]**

Bemerkungen:
keine

- PMAP

Umrechnen der physikalischen Koordinaten des Bildschirms in globale Koordinaten und umgekehrt.

Syntax: **v = PMAP(x,n)**

Bemerkungen:

Solange die Anweisung WINDOW nicht gegeben wird, sind "globale" und "physikalische" Koordinaten des Grafikbildschirms indentisch - (0,0) bezeichnet das Pixel in der oberen linken Ecke, (0,200) das Pixel in der unteren linken Ecke usw. Mit WINDOW lassen sich diese Zuordnungen nicht nur in ihrer Folge, sondern auch in ihrem Wertebereich veraendern.

Die Anweisung

```
WINDOW SCREEN (-500,500) - (500,-500)
```

legt fest: Das Pixel in der oberen linken Ecke bekommt die "globalen" Koordinaten (-500,500) zugewiesen, das Pixel in der unteren linken Ecke die Koordinaten (500,-500) usw. Folgende Zeichenbefehle wie PSET, LINE und CIRCLE arbeiten mit "globalen" Koordinaten.

Mit PMAP laesst sich ermitteln, welches physikalische Pixel einer globalen Koordinate zugeordnet ist und umgekehrt.

4.6.3.2. Funktionen

- POINT

Uebergeben des Attribut-Wertes eines Punktes oder der Position des Kursors auf dem Grafikbildschirm.

Syntax: `v = POINT (x,y)`
`v = POINT (n)`

Die erste Form liefert den Attribut-Wert (Farbe) des Pixels auf den angegebenen Koordinaten (x,y). Die Koordinaten koennen in absoluter oder relativer Form angegeben werden.

- SCREEN

Uebergeben des ASCII-Codes des Zeichens an der angegebenen Position.

Syntax: `v = SCREEN (Zeile, Spalte)[,z]`

Bemerkungen:
keine

4.7. Musik

4.7.1. Anweisungen

- SOUND

Erzeugen eines Tones ueber den Lautsprecher.

Syntax: `SOUND Freq,Dauer`

Bemerkungen:
keine

- PLAY

Erzeugen von Toenen mit Hilfe einer Zeichenkette.

Syntax: `PLAY Zeichenkette`

Bemerkungen:

PLAY besitzt eine Tondefinitionssprache. Eine Serie von Toenen wird als Zeichenfolge definiert - PLAY spielt sie unabhaengig vom restlichen Verlauf des Programms ab. Die maximal zulaessige Laenge von **Zeichenkette** (theoretisch 32767 Zeichen) ist nur im **"Vordergrundbetrieb"** nutzbar. Im **"Hintergrund-betrieb"** ist sie auf die Groesse des Musik-Puffers (maximal 4096 Noten) begrenzt.
TBASIC stellt als Standardvorgabe einen Puffer fuer 32 "Hintergrundnoten" zur Verfuegung, der mit dem Compiler-Befehl **$SOUND** bis auf 4096 Noten vergroessert werden kann.

Ueber PLAY ON und ON PLAY kann ein Unterprogramm festgelegt werden, das periodisch den Zustand des Notenpuffers prueft und ihn wenn notwendig mit neuen PLAY-Befehlen fuellt.

Hinweis:

Werden mehr Befehle und Noten angegeben, als in den (durch $SOUND gesetzten) Musik-Puffer hineingehen, dann arbeitet PLAY die ueberzaehligen Befehle im Vordergrundbetrieb ab.

Jeder PLAY-Befehl kann entweder in Form einer Konstanten oder mit **=VARPTR$(Variable)** gegeben werden, wobei Variable einen numerischen Wert enthalten muss.

Beachte:
Im BASIC-Interpreter BASI ist die Angabe von Variablennamen innerhalb der Befehlszeichenkette von PLAY erlaubt (z.B. "0=X"). In TBASIC ist diese Form nicht moeglich, da hier Variablennamen nur waehrend der Compilierung verwendet werden. Deshalb muss hier unbedingt die Form

```
"0 = " + VARPTR$(X)
```

verwendet werden.

4.7.2. Anweisungen zur Programmunterbrechung

- PLAY ON/PLAY OFF/PLAY STOP

Syntax:
```
PLAY ON
PLAY OFF
PLAY STOP
```

Bemerkungen:
keine

- ON PLAY(n)

Spielen von Musik waehrend der Programmunterbrechung.

Syntax: **ON PLAY(n) GOSUB Label**

Bemerkungen:

Label bezeichnet den Startpunkt eines Unterprogramms.

Wie auch alle anderen ON...-Anweisungen legt ON PLAY lediglich eine Zuordnung fest. Eine sichtbare Reaktion von TBASIC auf ON PLAY findet erst dann statt, wenn das Abspielen von Musik mit PLAY ON gestartet wird. Nach diesem Befehl wird vor der Ausfuehrung jedes Programmschritts geprueft, ob der Puffer fuer PLAY noch mehr Noten enthaelt, als durch **n** festgelegt ist. Wenn ja, wird das Programm fortgesetzt, ansonsten wird das durch **Label** bezeicnete Unterprogramm aufgerufen. Mit PLAY OFF laesst sich diese Ausfuehrung wieder abschalten.

Mit dem Compiler-Befehl **$EVENT** kann festgelegt werden, welche Teile des Programms durch ON PLAY bzw. PLAY ON unterbrechbar sind.

4.7.3. Funktionen

- PLAY

Uebergeben der Anzahl Noten, die sich noch im Puffer fuer Hintergrundmusik befinden.

Syntax: **v = PLAY(n)**

Bemerkungen:
keine

4.8. Prozessorarbeit

4.8.1. Anweisungen und Funktionen fuer den Zugriff auf den Speicher

- DEF SEG

Legt fest, auf welches Speichersegment sich nachfolgende Adressangaben fuer **BLOAD, BSAVE, CALL ABSOLUTE, PEEK** und **POKE** beziehen.

Syntax: **DEF SEG [=Segment]**

Bemerkungen:

Segment ist eine vorzeichenlose Integerzahl im Bereich von 0 bis 65535 und kann auch eine Konstante mit einer anderen Basis als 10 sein - in den meisten Faellen wird hier eine Hexadezimalzahl verwendet.

Eine absolute Speicheradresse (Bereich von 0...1024 KByte) wird in zwei voneinander getrennten Werten ausgedrueckt - dem **Relativzeiger**, einem Wert im Bereich von 0 bis 64 KByte und einem **Segment**. Der Wert fuer das Segment legt fest, welcher der 16 moeglichen 64 KByte-Bereiche des gesamten Speichers gemeint ist. Beide Werte sind Integerzahlen im Bereich von 0 bis 65535 (16 Bit), die absolute Speicheradresse ergibt sich aus

Segmentadresse * 16 % Relativzeiger

DEF SEG ohne weitere Argumente uebergibt die Adresse des Datensegments, dessen erste 256 Byte die Systemvariablen der Laufzeitbibliothek enthalten. Die Speicherzellen 0 und 1 dieses Segments enthalten die Segmentadresse des Zeichenkettenspeicherbereichs.

- POKE

Schreibt ein Byte in eine Speicherzelle.

Syntax: **POKE Adresse, Wert**

Bemerkungen:

Adresse bezeichnet die Adresse innerhalb des zuletzt mit DEF SEG gesetzten Speichersegments.

Wert ist der Inhalt, der in die angegebene Speicherzelle geschrieben werden soll.

- PEEK

Uebergibt ein gelesenes Byte von der angegebenen Speicherzelle.

Syntax: **v = PEEK (Adresse)**

Bemerkungen:

Adresse bezeichnet die zu lesende Speicherzelle innerhalb des Segments, das mit der letzten DEF SEG-Anweisung gesetzt wurde.

- BSAVE

Der Inhalt eines angegebenen Speicherbereiches wird als Datei auf die Diskette kopiert.

Syntax: **BSAVE Dateiname, Relativzeiger, Laenge**

Bemerkungen:

Wenn die durch **"Dateiname"** bezeichnete Datei noch nicht existiert, wird sie erzeugt. Existiert sie dagegen bereits, wird ihr alter Inhalt ueberschrieben.
Mit BSAVE erzeugte Dateien koennen mit BLOAD wieder in den Speicher geladen werden, wobei es moeglich ist, eine andere Startadresse anzugeben.
Vor dem Ausfuehren von BSAVE muss das **"momentane Segment"** explizit mit DEF SEG gesetzt werden.

Grenzen:
Sollte eine EGA-Karte im Computer benutzt werden, kann auf den Bildspeicher dieser Karte mit PEEK, POKE, BLOAD und BSAVE nicht direkt zugegriffen werden. Das direkte Abspeichern von Grafikbildern, die in dem Videomode 7...10 erstellt worden sind, ist also nicht moeglich.

- BLOAD

Der Inhalt einer mit BSAVE erzeugten Datei wird in den Speicher geladen.

Syntax: **BLOAD Dateiname[,Relativzeiger]**

Bemerkungen:

Der Zielbereich von BLOAD wird durch TBASIC nicht geprueft. Bei der Adressangabe ist deshalb zu beachten, dass nicht Teile von DCP oder vom Programm ueberschrieben werden.

Als Standardvorgabe fuer das Segment von BLOAD wird das Segment der einfachen Variablen angenommen. Wenn vergessen wird, den Zielbereich explizit mit DEF SEG zu setzen, wird die Datei in den Variablenbereich hineingeladen.

Grenzen:
Sollte eine EGA-Karte im Computer eingesetzt werden, kann auf den Speicher dieser Karte nicht mit BLOAD oder BSAVE zugegriffen werden. Ein direktes Laden von Grafikbildern, die im Videomode 7...10 (siehe SCREEN) erzeugt worden sind, ist mit BLOAD nicht moeglich.

- REG

Lesen bzw. Setzen der Inhalte des Puffers fuer die Prozessor-Register.

Syntax: **Anweisung: REG Register, Wert**
Funktion: v = REG(Register)

Bemerkungen:

Register ist ein Integer-Ausdruck. Sein Wert bezeichnet eines der Register des Prozessors, dessen Inhalt im Puffer REG gespeichert ist.

Wert ist ein Integer-Ausdruck im Bereich von 0 bis 65535 (16 Bit).

Dieser Puffer enthaelt die Registerinhalte des Prozessors nach Ausfuehrung der jeweils letzten (mit CALL ABSOLUTE oder CALL INTERRUPT aufgerufenen) Maschinensprache-Routine. Er kann als Feld mit 10 Elementen vom Typ Integer aufgefasst werden:

REG-Nummer	= Prozessor-Register
0	Flag-Register
1	AX
2	BX
3	CX
4	DX
5	SI
6	DI
7	BP
8	DS
9	ES

Wenn der Wert von **Register** nicht im Bereich von 0 bis 9 liegt, erzeugt REG den Laufzeitfehler 5 (Illegaler Funktionsaufruf).

REG als **Anweisung** erwartet zwei Parameter und setzt ein Element des Puffers:

```
REG 2, &HB100     'setzt Register BX auf &HB100
```

REG als **Funktion** erwartet als Argument die Nummer des Registers und liefert einen Integerwert zurueck:

```
PRINT REG(2)      'gibt den Inhalt von BX aus
```

- VARPTR

Uebergibt die Adresse einer Variablen.

Syntax: **v = VARPTR(Variable)**

Bemerkungen:

Es wird die Adresse ermittelt, auf der das erste Byte des Variableninhalts gespeichert ist.
Da TBASIC den ganzen verfuegbaren Speicher (maximal 640 KByte) nutzt, ermittelt VARPTR nur einen Teil der tatsaechlichen Adresse, naemlich den Relativzeiger innerhalb eines Speichersegments. In welchem Speichersegment sich die Variable befindet, wird mit der Funktion VARSEG ermittelt.

Eine komplette Adressermittlung besteht also aus zwei Schritten:
Ermittlung der vollstaendigen Adresse der ZK-Variablen X$:

```
- Segment% = VARSEG(X$)
- Zeiger%  = VARPTR(X$)
```

Numerische Felder enthalten sofort ihre Werte. Die Elemente von Zeichenkettenfeldern enthalten dagegen einen Zeiger und eine Laengenangabe.

Beispiel:

siehe VARSEG

- VARPTR$

Uebergibt den Zeiger zu einer Variablen in Form einer Zeichenkette.

Syntax: **v$ = VARPTR$(Variable)**

Bemerkungen:

Variable ist eine beliebige numerische oder Zeichenkettenvariable oder ein Element eines Feldes.

VARPTR$ uebergibt einen vollstaendigen Zeiger zu einer Variablen in Form einer Zeichenkette und wird eigentlich nur zusammen mit PLAY oder DRAW benutzt, um die Werte von Variablen mit in eine Befehls-Zeichenkette aufzunehmen. Die Zeichenkette hat eine Laenge von 5 Byte, die letzten vier beinhalten die Variablenadresse in der Form Segment:Relativzeiger.

Beispiel:

```
SCREEN 1          'Grafik 320*200

HAUS$ = "C20 G5 H20 F20 H5 D20 L30"
DRAW HAUS$

DRAW "BZ3":DRAW "P1,3"

FOR I% = 1 TO 280 STEP 40
    DRAW "BM=" + VARPTR$(I%) + ",40"
    DRAW HAUS$
NEXT I%

END
```

- VARSEG

Uebergibt die Segmentadresse einer Variablen.

Syntax: **v = VARSEG(Variable)**

Bemerkungen:

Variable ist der Name einer beliebigen numerischen Zeichenkettenvariablen oder die Bezeichnung eines Feldelements.

Jede (absolute) Adresse im Speicher setzt sich aus zwei Teilangaben zusammen: einer Segmentadresse und einem Relativzeiger innerhalb des Segments (siehe DEF SEG).

Um die Adresse einer Variablen (z.B. fuer die Uebergabe ihres Inhalts an eine Assembler-Routine) vollstaendig zu bestimmen, werden sowohl die Funktion VARPTR als auch die Funktion VARSEG benoetigt.
VARSEG uebergibt einen Integer-Wert im Bereich von 0 bis 65535, der angibt, in welchem Speichersegment sich die Variable befindet.

TBASIC speichert alle "einfachen" Variablen zusammen in einem Segment, maximal koennen also 64 KByte fuer einfache Variablen belegt werden. Aus diesem Grund liefert VARSEG fuer alle einfachen Variablen denselben Wert zurueck. (Dieses Segment wird auch von PEEK und POKE bearbeitet, falls vergessen wurde, mit DEF SEG eine Segmentadresse zu setzen.)

Da jedes Feld fuer sich allein eine maximale Groesse von 64 KByte haben kann, sind hier auch die Segmentadressen verschieden - VARSEG uebergibt jeweils einen anderen Wert, wie das folgende Beispielprogramm beweist:

Beispiele:

```
DIM FELD1%(20), FELD2(20), FELD$(40)
X%=1: Y=2: Z=3      '"einfache" Variablen

'Ausgabe der absoluten Adressen der Felder:

PRINT "FELD1%: $"; HEX$(VARSEG(FELD1%(0)))_
      ":"; HEX$(VARPTR(FELD1%(0)))
PRINT "FELD2: $"; HEX$(VARSEG(FELD2(0)))_
      ":"; HEX$(VARPTR(FELD2(0)))
PRINT "FELD$: $"; HEX$(VARSEG(FELD$(0)))_
      ":"; HEX$(VARPTR(FELD$(0)))

'Ausgabe der absoluten Adressen der
'"einfachen" Variablen:

PRINT "Variable X%: $"; HEX$(VARSEG(X%))_
      ":"; HEX$(VARPTR(X%))
PRINT "Variable Y: $"; HEX$(VARSEG(Y))_
      ":"; HEX$(VARPTR(Y))
PRINT "Variable Z: $"; HEX(VARSEG(Z))_
      ":"; HEX$(VARPTR(Z))
```

In einem numerischen Feld sind die Werte direkt gespeichert und koennen ueber Zugriffe auf die mit VARSEG und VARPTR ermittelte Adresse gelesen werden.
Bei Zeichenkettenfeldern stehen innerhalb des Feldes selbst zwei Werte (16 Bit): die Adresse innerhalb des Zeichenketten-Segments und die momentane Laenge der Zeichenkette. Um den Zeichenketteninhalt mit PEEK zu lesen, sind folgende Schritte erforderlich:

```
DIM A$(20): A$(1) = "Inhalt"
DEFINT A-Z

'Lesen von Startadresse/Laenge:
DEF SEG = VARSEG(A$(0))    'Segment auf Feld
SX = VARPTR(A$(0))         'Adresse Element A$(0)

ZKStart = PEEK(SX) + 256*PEEK(SX+1)
ZKLen = PEEK(SX+2)+256*PEEK(SX+3)
PRINT "Startadresse des Inhalts:" ZKStart
PRINT "Zeichenketten-Laenge:" ZKLen

DEF SEG                        'System-Segment setzen
ZKSeg = PEEK(0) + 256*PEEK(1)    '$-Segment
DEF SEG = ZKSeg                'ermitteln und setzen

'Tatsaechlich: Hier ist der Zeichenketteninhalt..
FOR X = 0 TO ZKLen-1
    PRINT CHR$(PEEK(ZKStart+X));
NEXT
```

- OUT

Sendet ein Byte (8 Bit) zu einem I/O-Port.

Syntax: **OUT n,m**

Bemerkungen:
keine

- INP

Ubergibt ein gelesenes Byte vom Port n.

Syntax: **v = INP(n)**

Bemerkungen:
keine

- WAIT

Wartet auf einen bestimmten Wert, der von einem ausgewaehlten Port gelesen wird.

Syntax: **WAIT Port, Bitmuster, Maske**

Bemerkungen:

WAIT prueft jeweils ein Byte.
WAIT haelt die Ausfuehrung des Programms solange an, bis ein bestimmter Wert von dem durch Port angegebenen I/O-Port gelesen wird. Die Parameter **Bitmuster** und **Maske** erlauben eine Pruefung des gelesenen Wertes auf fast alle erdenklichen Arten.

Der gelesene Wert wird zwei Operationen unterzogen:

1. **XOR** mit dem fuer **Maske** angegebenen Wert. Wenn **Maske** nicht angegeben ist, benutzt TBASIC den Wert &H0000 0000, laesst den gelesenen Wert also unveraendert. Durch **"eine Eins"** im Wert von Maske wird das korrespondierende Bit des gelesenen Wertes umgedreht, also z.B.

```
1100 0011      (gelesener Wert)
0000 1111      (Maske)
---------
1100 1100      (Ergebnis)
```

2. **AND** mit dem fuer **Bitmuster** angegebenen Wert. Wenn das Gesamtergebnis den Wert 0 hat, wiederholt TBASIC die Leseaktion des Ports solange, bis die beiden Operationen einen Wert ungleich 0 ergeben. Ein Beispiel zur AND-Operation:

```
1100 1100      (Ergebnis nach XOR)
0000 0100      (Bitmuster)
---------
0000 0100
```

Anders gesagt: Eine **AND**-Operation ohne vorheriges **XOR** prueft auf gesetzte Bits; ein vorheriges **XOR** dreht gelesene Bits um, die nachfolgende **AND**-Operation prueft auf geloeschte Bits.

Beispiel:

```
WAIT &H60,1          'Tastatur-Status direkt
                     '(wartet auf einen Tastendruck)
PRINT "Hallo!"
```

- ENDMEM

Uebergibt die Adresse der hoechsten zur Verfuegung stehenden Speicherzelle.

Syntax: **v = ENDMEM**

Bemerkungen:

Das Ergebnis wird im Format Langinteger uebergeben und sollte wegen moeglicher Rundungsfehler auch einer Langinteger-Variablen zugewiesen werden. Nach Bestimmen der hoechsten verfuegbaren Adresse kann ueber die Anweisung **MEMSET** ein Speicherbereich **"oberhalb"** des Programms fuer Unterprogramme in Maschinensprache reserviert werden.

Der von ENDMEM uebergebene Wert bleibt waehrend der Programmausfuehrung konstant. Er wird durch ein Herabsetzen der Obergrenze mit MEMSET nicht veraendert.

Grenzen:
ENDMEM fuehrt keine eigene Pruefung aus, sondern benutzt einen von DCP gelieferten Wert. Dieser Wert muss (speziell nach der Installation von RAMDisks) nicht unbedingt mit dem physikalischen Ende des Speichers identisch sein.

Beispiel:

```
PRINT "RAM-Ende bei: "; ENDMEM
```

- MEMSET

Legt die obere Grenze des von TBASIC benutzten Speicherbereichs fest.

Syntax: **MEMSET Adresse**

Bemerkungen:

Adresse ist ein Langinteger-Ausdruck, der die Obergrenze des nutzbaren Speichers absolut festlegt.

Damit wird ein Teil des Speichers, der von TBASIC nicht benoetigt wird, zur Speicherung von Maschinenprogrammen reserviert.

In der Praxis wird MEMSET zusammen mit der Funktion ENDMEM und einer Konstanten benutzt (siehe Beispiel).

Der Versuch, die Obergrenze des Speichers mit MEMSET soweit herunterzusetzen, dass fuer bereits gespeicherte Daten bzw. das Programm nicht mehr genuegend Platz bleibt, hat den Laufzeitfehler "Nicht genug Speicherplatz" zur Folge. Eine vorherige Pruefung des freien Speicherplatzes mit **FRE(-1)** erspart unangenehme Ueberraschungen dieser Art.

Beispiel:

```
'Ausgabe des freien Speicherplatzes fuer Felder
PRINT FRE(-1)

'Reservieren der obersten 128 Byte des zur
'Verfuegung stehenden Speichers:
Obergrenze& = ENDMEM
MEMSET Obergrenze&-128

'dasselbe Ergebnis bringt: MEMSET ENDMEM-128

PRINT FRE(-1)     'es werden 128 Byte weniger
                  'ermittelt

'An dieser Stelle koennte man jetzt mit POKE oder
'BLOAD eine Assembler-Routine einschreiben:
DEF SEG = ENDMEM/65536    'Segment-Adresse

'   BLOAD "xxx.obj", 0     'Laden einer Datei

'   FOR X% = 0 TO 127      'oder eintragen mit POKE
'       READ A%: POKE X%,A%
'   NEXT
'   DATA 1,2,3,4,5,6,7,8,.....

'Rueckgabe des Speicherplatzes:

MEMSET ENDMEM    'wieder auf Obergrenze
PRINT FRE(-1)
END
```

4.8.2. Anweisungen fuer die Unterprogrammarbeit

- CALL

Es wird eine benutzerdefinierte Prozedur aufgerufen.

Syntax: **CALL Prozedurname [(Parameterliste)]**

Bemerkungen:

Prozedurname ist der Name einer Prozedur, die an einem beliebigen Punkt des Programms (mit SUB) definiert wurde.

Parameterliste ist eine (optionale) Folge von Variablennamen, Ausdruecken und/oder Konstanten, deren Werte bzw. Adressen der Prozedur uebergeben werden.

Anzahl, Reihenfolge und Typ der mit CALL uebergebenen Parameter muessen mit der bei der Definition der Prozedur angegebenen Parameterliste uebereinstimmen, solange die Prozedur nicht als **INLINE** definiert ist - ansonsten wird bereits waehrend des Compilierens mit einer Fehlermeldung abgebrochen.

Bei Prozeduren vom Typ **INLINE** nimmt der Compiler keine Pruefung vor. Fuer die Bestimmung der Parameterzahl und des Typs der einzelnen Parameter ist die Prozedur selbst verantwortlich (siehe Kapitel 5).

Als Adressen uebergebene Variablen koennen durch die aufgerufene Prozedur veraendert werden. In Klammern eingeschlossene Variablen sowie Konstanten und Ausdruecke werden grundsaetzlich als Werte uebergeben, Veraenderungen innerhalb der Prozedur haben eine rein lokale Wirkung.

Felder werden bei einer Parameteruebergabe durch Anhaengen eines leeren Klammerpaars an den Namen als solche gekennzeichnet und grundsaetzlich als Adresse uebergeben:

```
CALL Summe (A())      'uebergibt Feld A
CALL Summe (A(3))     'uebergibt Element A(3)
```

Beispiel:

```
'Diese Prozedur gibt lediglich die Werte
'aller ihr uebergebenen Parameter aus. A(1)
'bezeichnet ein eindimensionales Feld.

SUB TestProc (I% L&, S!, D#, E, A(1))
    PRINT I%, L&, S!, D#, E, A(0)
END SUB                'Prozedurende

DIM Array(1)           'zwei numerische Elemente
Integer% = 1: Langinteger& = 2
Einfachgenau! = 3: Doppeltgenau# = 4
Array(0) = 5

CALL TestProc (Integer%, Langinteger&,_
     Einfachgenau!, Doppeltgenau#,_
     Integer% ^2, Array())
```

- CALL ABSOLUTE

Es erfogt ein absoluter Aufruf einer in Assemblersprache geschriebenen Routine.

Syntax: **CALL ABSOLUTE Adresse (Parameterliste)**

Bemerkungen:

Adresse ist ein Integer-Ausdruck im Bereich von -32768 bis 65535. Dieser bezeichnet den Relativzeiger (16 Bit) innerhalb des mit der letzten DEF SEG-Anweisung gesetzten Segmentes.

Parameterliste ist der Inhalt des Registerpuffers.

Ueber den Compiler-Befehl **$INLINE** ist das Einbinden von Maschinenprogrammen erheblich einfacher - CALL ABSOLUTE sollte deshalb nur dann eingesetzt werden, wenn ein existierendes BASIC-Programm uebernommen und fuer TBASIC modifiziert wird.

TBASIC uebergibt nicht wie bei BASI eine Argumentenliste, sondern einen **Registerpuffer.**
Der Inhalt des Registerpuffers wird bei Ausfuehren eines CALL ABSOLUTE oder CALL INTERRUPT in die einzelnen Register des Prozessors (ausser SS und SP) geladen. Direkt nach dem Ruecksprung der Assembler-Routine werden die Registerinhalte wieder im Registerpuffer gespeichert.

Dieser Puffer enthaelt damit immer die Registerinhalte des Prozessors nach Ausfuehrung der jeweils letzten Assembler-Routine.
Der Puffer kann als Feld mit 10 Elementen vom Typ Integer aufgefasst werden:

REG-Nummer	**= Prozessor-Register**
0	Flag-Register
1	AX
2	BX
3	CX
4	DX
5	SI
6	DI
7	BP
8	DS
9	ES

Der Zugriff auf einzelne Elemente dieses Puffers (d.h. einzelne Register) geschieht ueber das reservierte Wort REG, das entweder eine Anweisung oder eine Funktion darstellt.

REG als **Anweisung** erwartet zwei Parameter und setzt ein Element des Puffers:

```
REG 2, &HB100          'setzt Register BX auf &HB100
```

REG als **Funktion** erwartet als Argument die Nummer des Registers und liefert einen Integerwert zurueck:

```
PRINT REG(2)           'gibt den Inhalt von BX aus
```

Eine mit CALL ABSOLUTE aufgerufene Routine muss mit einem **"langen"** RET-Befehl (RET FAR) enden. Die Programmausfuehrung wird danach mit dem Befehl fortgesetzt, der sich unmittelbar hinter CALL ABSOLUTE befindet.

Grenzen:
Der Programmierer bzw. die mit CALL ABSOLUTE aufgerufene Routine ist selbst dafuer verantwortlich, dass die Inhalte der Prozessor-Register DS (Datensegment), BP (Base Pointer), SS und SP (Stacksegment, Stackpointer) nicht veraendert werden.

Beispiel:

Ein Unterprogramm in Assemblersprache kann beispielsweise dazu benutzt werden, um Elemente numerischer Felder auf einen bestimmten Wert zu setzen. Das folgende Programm demonstriert eine Routine, die drei Parameter mit REG uebergeben bekommt: die Startadresse des Feldes, die Anzahl der Elemente und den einzusetzenden Wert.

Um ein Assemblerprogramm fuer CALL ABSOLUTE in den Speicher zu bringen, gibt es zwei Moeglichkeiten: Entweder wird ein Bereich oberhalb des Programms mit MEMSET reserviert und das Programm mit POKE in diesen Bereich geschrieben oder man verwendet einfach ein Integer-Feld und belegt dessen Elemente mit den einzelnen Maschinenbefehlen. Diese Methode funktioniert natuerlich nur, wenn das Programm keine absoluten Referenzen auf sich selbst enthaelt.

```
'Definition eines Integer-Feldes, in dem das
'Maschinenprogramm "versteckt" wird:

DIM HX%(10)             '22 Byte
HX%(0) = &HF2FC         'CLD / REPNZ
HX%(1) = &HCBAA....     'STOSW / RET FAR

'ein Feld, dessen Elemente alle auf einen
'bestimmten Wert gesetzt werden sollen:

DIM Feld%(100)          '101 Integer-Elemente

'Setzen der Prozessor-Register fuer den Aufruf:
REG 1, 55               'AX: der einzusetzende Wert
REG 3, 101*2            'CX: 101 Wiederholungen
REG 6, VARPTR(Feld%(0))  'SI: Zieladresse
REG 9, VARSEG(Feld%(0))  'ES: Ziel-Segment

'Aufruf der Assemblerroutine: Setzen des
'Segments (d.h. der Startadresse):
DEF SEG = VARSEG(HX%(0))
CALL ABSOLUTE (VARPTR(HX%(0))

'Hat es funktioniert?
FOR X% = 0 TO 100
    PRINT Feld%(X%));
NEXT X%

END
```

- CALL INTERRUPT

Aufruf eines System-Interrupts.

Syntax: **CALL INTERRUPT n**

Bemerkungen:

n ist ein Integer-Ausdruck im Bereich von 0...255 und steht fuer die Nummer des aufzurufenden Interrupts.

Vor der Ausfuehrung von CALL INTERRUPT werden die Prozessor-Register mit dem Inhalt des Registerpuffers geladen, direkt nach der Ausfuehrung werden die Prozessor-Register in diesen Puffer zurueckgespeichert. Mit REG kann auf die einzelnen Registerinhalte zugegriffen werden (siehe CALL ABSOLUTE).

Die DCP-Dokumentation enthaelt eine komplette Beschreibung saemtlicher zur Verfuegung stehender Interrupts sowie der erforderlichen Registerinhalte beim Aufruf bzw. der Bedeutung zurueckgelieferter Werte.

Bei selbstgeschriebenen Interrupt-Routinen muss strengstens darauf geachtet werden, dass die Inhalte der Prozessor-Register SS und SP (Stacksegment und Stackpointer) nicht veraendert werden.

Grenzen:
Vor einem Aufruf mit CALL INTERRUPT muessen die entsprechenden Register des Prozessors (d.h. die Elemente des Registerpuffers) korrekt gesetzt sein - ansonsten sind die Ergebnisse unvorhersagbar.

Beispiel:

```
'Ausgabe des Zeichens "A" ueber einen
'direkten Aufruf der DCP-Funktion 02

'Register 1H auf 02: Funktionsnummer
REG 1, &H0200

'Register DL auf &H41: Zeichen "A",
'Register DH bleibt unveraendert
REG 4, REG(4) AND &HFF00 OR &H0041

CALL INTERRUPT &H21          'DCP-Aufruf
```

4.9. DCP-typische BASIC-Erweiterungen

4.9.1. Verzeichniszugriff

- MKDIR

Erstellt ein Verzeichnis (Subdirectory) auf der Diskette.

Syntax: **MKDIR Pfad**

Bemerkungen:
keine

- CHDIR

Ermoeglicht eine Aenderung des aktuellen Verzeichnisses.

Syntax: **CHDIR Pfad**

Bemerkungen:
keine

- RMDIR

Loescht das angegebene Verzeichnis von der Diskette.

Syntax: **RMDIR Pfad**

Bemerkungen:
keine

4.9.2. Laden von BASIC- und DCP-Programmen

- COMMAND$

Diese Funktion uebergibt eventuelle Zusatzparameter, die beim Start des Programms von der DCP-Kommandoebene mit angegeben wurden.

Syntax: **v$ = COMMAND$**

Bemerkungen:

Diese Funktion uebergibt den Inhalt der Zeile, die beim Start des Programms von der DCP-Kommandoebene mit eingegeben wurde. Lediglich der Programmname wurde automatisch entfernt.

COMMAND$ wird ueblicherweise fuer die Angabe weiterer Parameter zur Laufzeit benutzt wie z.B.

```
AUSDRUCK KAP1.TXT 2 7
```

AUSDRUCK ist in diesem Fall der Name eines in TBASIC geschriebenen Programms, das als .EXE-Datei compiliert wurde. **KAP1.TXT** bezeichnet den Namen der auszudruckenden Datei, die Zahlen 2 und 7 stehen fuer die erste und die letzte auszudruckende Seite. Das Programm AUSDRUCK benutzt die Funktion COMMAND$, um diese zusaetzlichen Parameter auszuwerten. COMMAND$ wuerde in unserem Beispiel die Zeichenkette KAP1.TXT 2 7 uebergeben, also die gesamte eingegebene Zeile mit Ausnahme des Programmnamens selbst.

Ein Programm, das mit COMMAND$ arbeitet, kann auch innerhalb der Programmierumgebung von TBASIC ausgetestet werden: COMMAND$ uebergibt hier die Zeichenfolge, die durch **Parameter line** im Menue **Options** gesetzt wurde.

Grenzen:
Die maximale Laenge einer DCP-Kommandozeile betraegt 127 Zeichen, die maximale Laenge uebergebener Zusatzparameter ist dadurch auf 127 minus der Laenge des Programmnamens begrenzt.

- SHELL

Von TBASIC aus wird ein anderes Programm geladen und ausgefuehrt. Anschliessend wird zu TBASIC zurueckverzweigt und das urspruengliche BASIC-Programm fortgesetzt.

Syntax: **SHELL [Befehlszeichenkette]**

Bemerkungen:

Befehlszeichenkette ist ein Zeichenketten-Ausdruck, der den Namen der .COM, .EXE- oder .BAT-Datei enthaelt, die ausgefuehrt werden soll. Hinter dem Namen der Datei koennen eventuelle Zusatzparameter folgen (siehe Beispiel).

Wird **Befehlszeichenkette** nicht angegeben, wird in die Kommandoebene von DCP verzweigt, d.h., COMMAND.COM wird allein in den Speicher geladen. An dieser Stelle koennen beliebige Kommandos (DIR, COPY...) eingegeben werden, der Befehl EXIT fuehrt zu einem Ruecksprung, und das BASIC-Programm kann fortgesetzt werden.

Beispiel:

```
PRINT "Dieses Programm benutzt den Befehl SHELL, "
PRINT "um das DCP-Kommando DIR/W auszufuehren "
PRINT "und so eine Datei-Liste auszugeben."
DELAY 3     'Zeit zum Lesen fuer den Benutzer

'"DIR" ist der Name des auszufuehrenden
'Kommandos. "/W" ist ein Zusatzparameter.
'Beide werden mit
'SHELL an COMMAND.COM uebergeben:

SHELL "DIR/W"     'fuehrt DIR/W unter DCP aus

DELAY 3
LOCATE 21,1
PRINT "Jetzt sind wir wieder in TBASIC."
END
```

4.9.3. Arbeit mit der BASIC-Umgebungstabelle

- ENVIRON

Aendert Parameter in der BASIC-Umgebungstabelle.

Syntax: **ENVIRON Zeichenketten-Ausdruck**

Bemerkungen:

Zeichenketten-Ausdruck besteht aus dem Namen des zu aendernden Parameters sowie den neu zu uebergebenden Daten und wird wie folgt angegeben:

"Parametername = [Daten][;]"

Mit ENVIRON und der verwandten Funktion ENVIRON$ koennen Parameter des Betriebssystems gesetzt werden, die auch fuer einen folgenden Aufruf von COMMAND.COM (mit SHELL) gueltig sind. Die Beschreibung der Kommandos SET im DCP-Handbuch enthaelt eine detaillierte Auflistung der einzelnen Moeglichkeiten.

Grenzen:
Der von der BASIC-Umgebungstabelle im Speicher belegte Platz kann mit ENVIRON nicht vergroessert werden. Vor dem Eintrag zusaetzlicher Parameter muss zuerst ein anderer Parameter geloescht werden.

Beispiel:

Der folgende Befehl setzt die Eintragung PFAD in der Tabelle auf die Zeichenkette \TBASIC. Nachfolgende Aufrufe von COMMAND.COM suchen automatisch dieses Verzeichnis ab, wenn sich die gesuchte Datei nicht innerhalb des Standard-Verzeichnisses befindet:

```
ENVIRON "PFAD=\TBASIC"
```

Um einen Parameter wieder aus der Tabelle zu loeschen, wird lediglich ein Semikolon nach dem Gleicheitszeichen angegeben:

```
ENVIRON "PFAD=;"
```

- ENVIRON$

Uebergibt die eingetragenen Parameter der BASIC-Umgebungstabelle.

Syntax: **v$ = ENVIRON$ (Parametername)**
v$ = ENVIRON$ (n)

Bemerkungen:

Parametername ist ein Zeichenketten-Ausdruck, mit dem der Name des zu lesenden Parameters angegeben wird.

n ist ein Integer-Ausdruck im Bereich von 1...255 und gibt die Nummer der zu lesenden Tabelleneintragung an.

Mit ENVIRON$ und der verwandten Anweisung ENVIRON koennen Parameter des Betriebssystems gesetzt werden, die auch fuer folgende Aufrufe von COMMAND.COM (mit SHELL) gueltig sind (siehe Beschreibung des Kommandos SET im DCP-Handbuch).

Wenn das Argument von ENVIRON$ ein Zeichenketten-Ausdruck ist, wird die TBASIC-Umgebungstabelle nach der angegebenen Zeichenfolge durchsucht und der unmittelbar darauffolgende Text innerhalb der Tabelle zurueckgeliefert. Wird der gesamte Eintrag nicht gefunden oder folgt kein weiterer Text in der Tabelle, dann liefert ENVIRON$ eine leere Zeichenkette zurueck. Es ist zu beachten, dass in diesem Fall zwischen Gross- und Kleinbuchstaben unterschieden wird.

Ist das Argument dagegen numerisch, dann uebergibt ENVIRON$ den n. Eintrag der Tabelle. Enthaelt die Tabelle weniger als n Eintraege, wird eine leere Zeichenkette zurueckgeliefert.

Beispiel:

Das folgende Programm gibt saemtliche momentan gesetzten Eintraege der Tabelle aus.

```
X% = 1
WHILE ENVIRON$(X%) <> ""
   PRINT ENVIRON$(X%)
   INCR X%
WEND
```

4.9.4. Verarbeitung von Steuerzeichen bei benutzereigenen Treibern

- IOCTL und IOCTL$

Kontrolle und Kommunikation mit dem Treiberprogramm des Peripheriegeraetes.

Syntax: **IOCTL [#]Dateinummer, Zeichenketten-Ausdruck**
v$ = IOCTL$([#]Dateinummer)

Bemerkungen:

Dateinummer ist ein Integer-Ausdruck und gibt die Dateinummer des entsprechenden Treiberprogramms an.

Zeichenketten-Ausdruck
enthaelt die Informationen, die dem Treiberprogramm uebergeben werden sollen.

Die von IOCTL$ uebergebene Zeichenkette enthaelt Daten des ueber **Dateinummer** adressierten Treiberprogramms. Das Format und die Art dieser Daten ist in sehr starkem Masse von der Art dieses Programms abhaengig.

4.9.5. Ermittlung von Fehlern bei Peripheriegeraeten

- ERDEV und ERDEV$

Es wird der Fehlerstatus eines Peripheriegerates bzw. des dazugehoerigen Treiberprogramms ermittelt.

Syntax: **v = ERDEV**
v$ = ERDEV$

Bemerkungen:

Wenn bei Operationen mit einem Peripheriegeraet ein Fehler auftritt, ruft DCP den internen **Interrupt 24** ("Critical error handler") auf, der Meldungen wie <A>bort<R>etry<I>gnore ausgibt.

Der dazugehoerige Vektor wird beim Start eines TBASIC-Programms so gesetzt, dass er auf eine Routine innerhalb der Laufzeitbibliothek zeigt - Meldungen wie die oben gezeigte erscheinen nicht auf dem Bildschirm. Wenn ON ERROR GOTO nicht gesetzt ist, haben sie direkt einen Laufzeitfehler zur Folge. Ist ON ERROR GOTO dagegen gesetzt, dann koennen die Art des Fehlers und das entsprechende Peripheriegeraet innerhalb der Fehlerbehandlungsroutine mit ERDEV und ERDEV$ bestimmt werden.

ERDEV uebergibt einen Integerwert, dessen niederwertiges Byte den Fehlercode enthaelt. Das hoeherwertige Byte enthaelt die Bits 15,14,13,3,2,1 und 0 des entsprechenden Peripheriegeraet-Statuswortes.

ERDEV$ enthaelt den Namen des Peripheriegeraetes, der bei zeichenorientierten Geraeten 8, bei blockorientierten Geraeten 2 Byte lang ist.

Eine Beschreibung der Geraetenamen, Statuswoerter und moegliche Fehlernummern ist dem DCP-Handbuch zu entnehmen.

Beispiel:

```
'Setzen der Fehlerbehandlungsroutine
ON ERROR GOTO Fehler

PRINT "Oeffnen Sie die Veriegelung des Laufwerks";_
      "A: und druecken Sie eine Taste...";

WHILE NOT INSTAT:WEND   'Warten auf Tastendruck
N$ = INKEY$

'Der Versuch, das Verzeichnis von A: zu lesen,
'endet mit einem Sprung zu Fehler:
FILES "A:\*.*"
END

Fehler:
  IF ERDEV = 2 AND ERDEV$ = "A:" THEN
     PRINT "Die Veriegelung ist offen!"
     PRINT "Bitte schliessen, Taste druecken..";
     WHILE NOT INSTAT: WEND: N$ = INKEY$
     RESUME     'erneuter Versuch!
  ELSE
     PRINT USING "Fehler ## auf Geraet &";_
           ERDEV, ERDEV$
  END IF
END             'Programmende
```

4.10. Compilerbefehle

Diese Befehle sind waehrend der Laufzeit des BASIC-Programms nicht erforderlich und gehoeren somit nicht zum eigentlichen Sprachumfang von TBASIC. Sie werden waehrend der Uebersetzung des BASIC-Programmes benoetigt und legen bestimmte Bedingungen fuer das Compilieren eines BASIC-Programms fest.
Zur Unterscheidung von "normalen" Befehlen beginnen sie grundsaetzlich mit einem **Dollarzeichen/Waehrungszeichen** und muessen stets allein auf einer Zeile des Quelltextes stehen. Sie werden im Gegensatz zu anderen BASIC-Compilern nicht innerhalb von Kommentaren versteckt.

4.10.1. Steuerbefehle

- $IF/$THEN/$ELSE/$ENDIF

Bedingtes Compilieren.

Syntax:

```
$IF Konstante
  .
  .                Anweisung(en)
  .
[$ELSE
  .
  .                Anweisung(en)
  .
    ]
$ENDIF
```

Bemerkungen:

Konstante steht hier fuer eine benannte Konstante oder einen angegebenen Wert. Wenn **Konstante** TRUE (ungleich 0) ist, dann uebersetzt der Compiler den Quelltext zwischen $IF und $ELSE (bzw. $ENDIF, falls $ELSE nicht angegeben ist); hat **Konstante** dagegen den Wert FALSE (0), dann wird der von $ELSE und $ENDIF umschlossene Quelltext uebersetzt (bzw. es passiert gar nichts, wenn $ELSE nicht angegeben wurde).

Mit diesen Compiler-Befehlen lassen sich ganze Programmteile beim Compilieren ausklammern bzw. einfuegen. Fuer die jeweilige Version wird einfach eine Konstante geaendert - und nicht ein ganzer Programmteil (siehe Beispiel).

$IF/$ELSE/$ENDIF-Konstrukte koennen bis zu einer Tiefe von 256 verschachtelt werden. Ein Programm kann beliebig viele bedingte Teile enthalten.

Das folgende Programm ist nur teilweise ausgefuehrt - es soll sowohl auf dem Computer mit einem Grafik-Adapter als auch auf einem Computer mit einem Monochrom-Adapter laufen.

```
CGA% = -1          '=TRUE
'                   Fuer Monochrom-Version CGA% = 0
$IF CGA%

    'Ein Programmteil, das die mittelhoch
    'aufloesende Grafik setzt und ein Programm in den
    'Speicher laedt

SCREEN 1            'Grafik 320x200
DEF SEG = &HB000
BLOAD "xxx.xxx"
DELAY 5
  .
  .
  .
$ELSE

'Bei Monochrom-Adapter wird nur
'der Bildschirm geloescht
CLS                 'loescht den Bildschirm
PRINT "..........."
                    '... usw.

$ENDIF

'ab hier geht es fuer beide Programmteile
'wieder gemeinsam weiter
```

4.10.2. Erzeugen von Feldern

- $STATIC

Setzt die Standardvorgabe des Compilers zum Erzeugen von Feldern auf "statisch".

Syntax: **$STATIC**

Bemerkungen:

In TBASIC ist das Erzeugen von Feldern auf zwei verschiedene Arten moeglich:

- **Statisch:**
  Der Compiler berechnet waehrend der Uebersetzung, wieviel Speicherplatz fuer ein Feld benoetigt wird. Er reserviert eine entsprechende Anzahl von Bytes. Statische Felder sind ein fester Bestandteil des Programms. Ein Loeschen mit **ERASE** setzt lediglich die einzelnen Feld-Elemente auf den Wert Ø bzw. eine Null-Zeichenkette zurueck. Der Zugriff auf einzelne Elemente geht schneller, weil die jeweiligen Adressen bereits waehrend des Compilierens berechnet werden.

- **Dynamisch:**
  Berechnen und Reservieren erfolgen zur Laufzeit des Programms. Der Befehl **ERASE** loescht ein dynamisches Feld komplett wieder aus dem Speicher und gibt den dazugehoerigen Speicherplatz wieder fuer andere Zwecke frei. Der Zugriff auf einzelne Elemente geht etwas langsamer, weil die jeweiligen Adressen zur Laufzeit des Programms berechnet werden muessen.

Der Compiler unterscheidet normalerweise anhand der Angabe fuer die Dimensionsgroesse(n), ob ein Feld statisch oder dynamisch erzeugt werden soll. Wenn die Groesse(n) durch Konstanten bestimmt werden, ist das erzeugte Feld statisch; werden sie ueber Variablen festgelegt, dann wird das Feld dynamisch erzeugt.

$STATIC ist die Standardvorgabe des Compilers und wird innerhalb des Quelltextes eigentlich nur dazu verwendet, um einen vorherigen $DYNAMIC-Befehl wieder rueckgaengig zu machen.

- $DYNAMIC

Setzt die Standardvorgabe des Compilers zum Erzeugen von Feldern auf "dynamisch".

Syntax: **$DYNAMIC**

Bemerkungen:

siehe Compiler-Befehl $STATIC

Die Standardvorgabe des Compilers ist $STATIC. Die Befehle $STATIC und $DYNAMIC koennen innerhalb eines Programms beliebig oft programmiert werden.

Wenn der Compiler-Befehl $DYNAMIC nicht vorliegt, erzeugen die ersten beiden der folgenden DIM-Befehle statische Felder.

```
DIM Feld1(20)          'Konstante als Groessenvorgabe
DIM STATIC Feld2(4)    'Zusatz STATIC
A = 10:DIM Test(A)     'Variable als Groessenangabe
DIM DYNAMIC C(10)      'Zusatz DYNAMIC
```

Nach dem Compiler-Befehl $DYNAMIC wuerde Feld1 als dynamisches Feld erzeugt, obwohl hier die Groesse ueber eine Konstante angegeben ist.

Beispiel:

```
'Freier Speicherplatz im Feldbereich:
PRINT "Freier Speicher: "; FRE(-1)
$DYNAMIC            'Standardvorgabe: dynamisch
DIM DTest(10000)    'wird dann dynamisch erzeugt
PRINT "Freier Speicher jetzt: ";FRE(-1)
$STATIC             'Standardvorgabe zurueck auf statisch
DIM STest(10000)    'wird statisch erzeugt
                    'und deshalb bereits vom
                    'Compiler belegt"
PRINT "Freier Speicher immer noch: ";FRE(-1)
ERASE STest         'setzt nur die Elemente zurueck
PRINT "Freier Speicher unveraendert: ";FRE(-1)
ERASE DTest         'loescht das dynamische Feld
                    'komplett
PRINT "Freier Speicher: ";FRE(-1)
```

4.10.3. Programmueberlagerung

- $INCLUDE

Einfuegen einer zusaetzlichen Quelltext-Datei waehrend des Uebersetzens.

Syntax: **$INCLUDE Dateinamen-Konstante**

Bemerkungen:

Dateinamen-Konstante steht fuer den Namen der Datei, die der Compiler lesen und an der Stelle des $INCLUDE-Befehls in den Quelltext einfuegen soll. Die Angabe des Dateinamens muss in der Form "x.x" erfolgen (also z.B. $INCLUDE "TEST.BAS"). Zeichenkettenvariablen sind zusammen mit $INCLUDE nicht erlaubt.

$INCLUDE ist ein zentraler Bestandteil von TBASIC. Der Befehl erlaubt es, sehr grosse Programme in mehrere Quelltext-Module zu unterteilen.
Dabei werden zwei verschiedene Arten von Dateien bearbeitet:

- eine Hauptdatei (Main file)
- einzelne Arbeitsdateien (Work file).

Der Main file enthaelt an einer oder an mehreren Stellen den Befehl $INCLUDE, gefolgt von dem Namen des Work files.
Beide Dateien muessen im Menue **Files**, Untermenue **Main file**, als entsprechende Datei angegeben werden.

Die Compilierung beginnt unabhaengig von dem momentanen Work file grundsaetzlich mit dem Main file. An der entsprechenden Stelle wird die mit $INCLUDE einzufuegende Datei eroeffnet und der von dort gelesene Quelltext uebersetzt. Wurde diese Datei komplett gelesen, geht die Compilierung mit dem naechsten Befehl nach $INCLUDE weiter.

$INCLUDE-Befehle koennen bis zu 15 Ebenen tief verschachtelt werden, d.h. eine als $INCLUDE geladene Datei kann ihrerseits weitere $INCLUDE-Befehle enthalten. Eine Datei kann beliebig viele $INCLUDE-Befehle derselben Ebene enthalten.

4.10.4. Musik

- $SOUND

Legt die Groesse des Puffers fuer PLAY fest.

Syntax: **$SOUND Puffergroesse**

Bemerkungen:

Puffergroesse ist eine Integer-Konstante und legt fest, wieviele Noten im Musik-Puffer gespeichert werden koennen. Zulaessige Werte gehen von 0 bis 4096.

Der Compiler-Befehl $SOUND dient als eine Initialisierungsbedingung und darf deshalb in einem Programm nur einmal gegeben werden.

Eine Note belegt 8 Byte im PLAY-Puffer. 4096 Noten wuerden also 32 KByte Speicherplatz entsprechen. Je groesser der Puffer ist, desto seltener muss ein Programm (mit ON PLAY GOSUB) den Musik-Puffer "nachladen".

Die Standardvorgabe von TBASIC ist 32 Noten. Sie kann ueber die Auswahl **Music buffer** (Menue **Options**, Untermenue **Metastatements**) neu festgesetzt werden. Ein $SOUND-Befehl im Quelltext setzt die Standardvorgabe ausser Kraft.

Die Auswahl **Music buffer** enthaelt ebenfalls ein Pruefen der Obergrenze. Durch entsprechende Eingaben laesst sich notfalls ausprobieren, ob mehr als 4096 Noten moeglich sind.

Es ist aber auch z.B. moeglich, eine PLAY-Anweisung mit 300 Noten zu geben, obwohl der Musik-Puffer zuvor auf eine Groesse von 200 Noten gesetzt wurde. In diesem Fall speichert PLAY soviele Noten wie moeglich in dem Puffer (hier: 200). Nicht speicherbare Noten werden im **Vordergrundbetrieb** abgespielt. Die Ausfuehrung des restlichen Programms wird entsprechend lange angehalten.

Beispiel:

```
$SOUND  200             'reserviert 1600 Byte fuer 200 Noten

'Die folgende PLAY-Anweisung spielt 250 "A"s. Die
'ersten 50 werden im Vordergrund bearbeitet,
'weil sie nicht in den Puffer hineinpassen:

PLAY "MB O2 L16" + STRING$(250, "A")

CLEAR:STOP              'wirkt erst nach Abspielen der
                        'ueberzaehligen 50 Noten
```

4.10.5. Datenfernverarbeitung

- $COM

Reserviert Speicherplatz fuer den Empfangspuffer eines seriellen Ports.

Syntax: **$COMn Groesse**

Bemerkungen:

n bezeichnet den jeweiligen seriellen Port (COM1 oder COM2).

Groesse ist eine Integer-Konstante und legt die Groesse des jeweiligen Puffers (in Byte) im Bereich 0... 32767 fest.
Angaben ausserhalb dieses Bereiches fuehren zu Fehlern beim Compilieren.

In einem Programm ist fuer jeden der beiden Ports nur jeweils ein $COM-Befehl zulaessig.

Die Standardvorgabe fuer beide Puffer ist eine Groesse von 256 Byte. Diese kann ueber das Menue **OPTIONS** (Untermenue **META-STATEMENTS**) geaendert werden.
Ein $COM-Befehl innerhalb des Quelltextes setzt die Vorgabe fuer den jeweiligen Port ausser Kraft.

Unterschiede:
Im BASIC-Interpreter BASI kann die Groesse der COM-Empfangspuffer nur ueber einen Kommandozeilen-Parameter gesetzt werden, sie gilt grundsaetzlich fuer beide Ports. TBASIC setzt die Groesse jedes Empfangspuffers einzeln und durch einen $COM-Befehl im Quelltext.

Das folgende Programm erzeugt natuerlich nur dann sichtbare Ergebnisse, wenn der Computer ueber COM1: auch Zeichen empfaengt.

```
'1024 Byte werden fuer den Empfangspuffer von COM1:
 reserviert:
$COM1 1024

'Festlegen einer "Abfangroutine" fuer empfangene
'Zeichen: Jeder Zeichenempfang ruft (unabhaengig
'vom restlichen Programmverlauf) die Routine
'COM1Eingabe auf:
ON COM(1) GOSUB COM1Eingabe

'und Aktivieren der Ueberwachung von COM1
COM(1) ON

'Zuordnen einer Dateivariablen zu COM1:
OPEN "COM1:" AS #1

'********** das "Hauptprogramm" **********

CLS
PRINT "Jeder Tastendruck --> Programmende"

WHILE NOT INSTAT            'solange kein Tastendruck
      LOCATE 4, 1
      PRINT TIME$           'Ausgabe der Uhrzeit
WEND

COM(1) OFF                  'Ueberwachung AUS
END                         'Programmende

'Bearbeitung empfangener Zeichen
COM1Eingabe:
    INPUT #1, X$            'liest das Zeichen
    LOCATE 8, 1
    PRINT "Zeichen empfangen:"; X$
RETURN
```

4.10.6. Unterbrechungssteuerung

- $EVENT

Kontrolliert das Erzeugen zusaetzlicher Codes fuer das Ueberwachen von Ports.

Syntax: **$EVENT ON/OFF**

Bemerkungen:

Wenn ein Programm eine oder mehrere **ON...GOSUB-Anweisungen** enthaelt, wird vor Ausfuehrung jedes Programmschrittes in Abhaengigkeit von einer Bedingung geprueft, ob das mit ON...GOSUB festgelegte Unterprogramm aufgerufen werden soll.

Ueber den Compiler-Befehl **$EVENT** kann festgelegt werden, welche Teile eines Programms durch mit **ON...GOSUB** festgelegte Hintergrundprozesse unterbrochen werden koennen - und welche nicht. Die Standardvorgabe des Compilers ist $EVENT ON, sobald der Quelltext einen oder mehrere ON...Anweisungen enthaelt.

Der Befehl $EVENT kann innerhalb eines Programms beliebig oft gegeben werden.

Beispiel:

```
'Setzen einer "Abfangroutine" fuer
'verschiedene Tasten:
ON KEY(1) GOSUB F1Taste
ON KEY(2) GOSUB F2Taste
ON KEY(3) GOSUB F3Taste

'und Aktivierung der Ueberwachung von F1...F3:
KEY(1) ON: KEY(2) ON: KEY(3) ON
'Der Compiler generiert ab hier zwischen jedem
'Programmschritt zusaetzlichen Pruefcode fuer
'Betaetigen von F1, F2 und F3. Das hat auch
'zeitliche Auswirkungen:

MTIMER                  'Mikrozaehler zuruecksetzen
FOR X% = 1 TO 10000
    Y% = 20
NEXT X%
Y% = MTIMER
PRINT "Zeit:"; Y%; "Mikrosekunden."

$EVENT OFF              'ab hier: keinen Pruefcode mehr
'erzeugen. Entspricht fuer unser Beispiel der
'Befehlsfolge: KEY(1) OFF: KEY(2) OFF: KEY(3) OFF

'und dieselbe Schleife noch einmal:
MTIMER                  'Mikrozaehler zuruecksetzen
FOR X% = 1 TO 10000
    Y% = 20
NEXT X%
Y% = MTIMER
PRINT "Zeit:"; Y%; "Mikrosekunden."

'ab hier: Pruefcode wieder an
$EVENT ON

'(weiterer Programmverlauf...)

END                     'Programmende

F1Taste:
 .
 .
 .
RETURN
```

```
F2Taste:
  .
  .
  .
RETURN

F3Taste:
  .
  .
  .
RETURN
```

4.10.7. Arbeit mit dem Speicher

- $STACK

Legt die Groesse des Stack fest, der dem Programm zur Verfuegung steht.

Syntax: **$STACK Groesse**

Bemerkungen:

Groesse ist eine Integer-Konstante im Bereich von 768 bis 32767 und legt die Groesse des Stack-Bereiches in Byte fest.

Der Stack des Prozessors wird vom Programm zum Speichern von Ruecksprungadressen (GOSUB, Funktions- und Prozeduraufrufe), zum Zwischenspeichern von Variablen innerhalb strukturierter Blocks und vor allem fuer lokale Variablen innerhalb von Prozeduren und Funktionen benutzt.

Der Befehl $STACK legt eine Initialisierungsbedingung fest und darf deshalb pro Programm nur einmal gegeben werden.

Die Standardvorgabe von TBASIC ist 768 (=&H300) Byte und stellt gleichzeitig das Minimum an Platz dar, der dem Stack mit $STACK zugeordnet werden kann. Die im Menue **Options** (Untermenue **Metastatements**) gesetzte Standardvorgabe wird durch einen $STACK-Befehl im Quelltext ausser Kraft gesetzt.

Um den Objektcode so kompakt wie moeglich zu halten, prueft TBASIC vor dem Aufruf eines Unterprogramms oder bei der Deklaration lokaler Variablen nicht, ob noch genuegend Platz auf dem Stack vorhanden ist. Wenn das Programm eine grosse Anzahl von Unterprogrammebenen enthaelt, lokale Felder oder rekursive Funktionsdefinitionen benutzt werden, dann sollte waehrend der Testphase grundsaetzlich der Schalter **Stack test** im Menue **Options** auf ON gesetzt werden.

Stack test prueft nicht nur, ob bei einem Aufruf genuegend Platz vorhanden ist, sondern auch, ob bei einem Ruecksprung eine entsprechende Adresse auf dem Stack liegt.

Ueberzaehlige RETURN-Anweisungen innerhalb eines Programms enden nur dann mit der Meldung **"RETURN ohne GOSUB"**, wenn ebenfalls der Schalter **Stack test** beim Compilieren auf **ON** gesetzt ist - ansonsten stuerzt das System ab.

Der noch freie Platz auf dem Stack kann jederzeit mit der Funktion **FRE(-2)** ueberprueft werden.

Beispiel:

```
'Reservierung von 8192 Byte fuer den Stack
$STACK 8192                'oder auch &H2000

PRINT FRE(-2)              'sollte rund 8150 ergeben
```

- $SEGMENT

Deklaration eines neuen Code-Segments.

Syntax: **$SEGMENT**

Bemerkungen:

Zum Erzeugen des kompaktesten Codes verwendet der Compiler normalerweise "kurze" Sprung- und Aufrufbefehle des Prozessors (jeder dieser Befehle belegt drei Byte im Speicher). Einen Nachteil haben "kurze" Befehle allerdings: Sie arbeiten maximal ueber einen Speicherbereich von 64 KByte hinweg. Das bedeutet, der maximale Offset betraegt 32767 in beiden Richtungen. Der Prozessor kann also mit einem Befehl maximal 32 KByte Code ueberspringen bzw. ein Unterprogramm aufrufen, das um diesen Betrag von der momentanen Stelle entfernt ist.

Wenn ein Programm so lang ist, dass der Prozessor diese Sprunggrenzen ueberschreiten muesste (theoretisch kann das bereits bei einem Programm mit 33 KByte Laenge passieren - naemlich dann, wenn auf der letzten Programmzeile ein Sprung zum Programmanfang steht), dann meldet sich der Compiler mit dem Fehler

```
Segment-Ueberlauf <ESC>-Taste
```

Fuer solche Faelle kommt der Compiler-Befehl $SEGMENT zur Anwendung. Damit kann festgelegt werden, wo ein neuer Segment-Bereich beginnt. **Innerhalb** dieses Bereiches werden nach wie vor "kurze" Spruenge verwendet; Spruenge und Aufrufe **zwischen** zwei Segmenten codiert der Compiler dagegen als "lang" (5 Byte pro Befehl). Die Ausfuehrung dieser Befehle dauert dementsprechend etwas laenger, zusaetzlich muessen bei Aufrufen einige Bytes mehr auf dem Stack gespeichert werden. Die negativen Begleiterscheinungen "langer" Spruenge lassen sich minimieren, wenn der Befehl $SEGMENT an natuerlichen Grenzen einzelner Programmteile gegeben wird (siehe Beispiel).

Ein Programm kann bis zu 16 verschiedene Segmente enthalten. Zu beachten ist, dass der Compiler nicht fuer jeden $SEGMENT-Befehl 64 KByte Speicherplatz reserviert. Er benutzt diese Unterteilung nur zum Unterscheiden zwischen "kurzen" und "langen" Spruengen. Nach der Uebersetzung gibt der Compiler im Fenster **Message** die Groessen einzelner Segmente durch Schraegstriche voneinander getrennt aus.

Hinweis:

Der Befehl $SEGMENT kann innerhalb einer Blockstruktur (FOR/NEXT, DO/LOOP, WHILE/WEND, SELECT oder IF/THEN/ELSE) nicht gegeben werden.

Beispiel:

Das folgende, nur teilweise ausgefuehrte Programm koennte die Grundstruktur eines Finanzbuchhaltungsprogramms sein. Es besteht aus einem Hauptmenue, aus dem eine der drei Hauptfunktionen ausgewaehlt werden kann. Jede dieser Hauptfunktionen ist in einem eigenen Segment gespeichert.

```
GOTO Hauptmenue                 'Sprung zum Hauptmenue

'Der erste Programmteil legt Kundenkonten an
Kunden:
   'Hier wuerde jetzt ein weiteres Menue stehen,
   'etwa mit Auswahlmoeglichkeitem wie
   '"bestehende Kosten auflisten", "Neues Konto
   'einrichten" usw. Die Auswahl "Zurueck" bringt
   'den Benutzer in das Hauptmenue zurueck.
  DO
   CALL KMenue                  '<-- innerhalb dieses Seg.:
   INPUT "Bitte waehlen Sie: "; Wahl$
   .
   .
   .
  LOOP UNTIL Wahl$ = "9"        'Auswahl "zurueck"
RETURN
  SUB KMenue
   .
   .
   .
   END SUB

$SEGMENT                        'ein neues Code-Segment
```

```
'Der zweite Programmteil  druckt Briefe,
'Mahnungen und Daten aus
Report:
     'Hier steht ebenfalls wieder ein Menue, die
     'letzte Funktion ist "Zurueck"
  DO
      CALL RMenue                 '<-- innerhalb dieses
                                  'Segments!
      CALL RWahl                  'dto
  LOOP UNTIL Wahl% = 9            '"Zurueck"
RETURN
  SUB RMenue
    .
    .
    .
  END SUB
  SUB RWahl
      INPUT Wahl%
    .
    .
    .
  END SUB

$SEGMENT      'ein weiteres Segment fuer Teil3

'Der dritte Programmteil erstellt eine Bilanz
Bilanz:
   'Anfrage nach dem Zeitraum usw., danach
   'Durchrechnung saemtlicher Konten
RETURN

$SEGMENT      'das vierte Segment fuer das
              'Hauptmenue

Hauptmenue:

  CALL Init             'Innerhalb dieses Segments...
  DO
                        'Ausgabe Hauptmenue
       INPUT "Bitte waehlen Sie: "; Haupt$
       'Nur die folgenden Aufrufe stellen "lange"
       'Spruenge dar - sie setzen das Programm
       'innerhalb eines anderen Segments fort!

   ON VAL(Haupt$) GOSUB Kunden, Report, Bilanz

   LOOP UNTIL Haupt$ = "9"          'Programmende
END

   SUB Init
     .
     .
     .
   END SUB
```

- $INLINE

Direktes Einfuegen von Maschinensprach-Befehlen in den Programmcode.

Syntax: **$INLINE Bytefolge| Dateinamen-Konstante**

Bemerkungen:

Bytefolge ist eine Folge von Integer-Konstanten mit einem zulaessigen Wertebereich von 0 bis 255, die voneinander durch Komma getrennt sind. Sie werden vom Compiler ohne weitere Interpretation an der Stelle des $INLINE-Befehls in den Objektcode aufgenommen.

Dateinamen-Konstante
ist der Name einer Datei, die ein Maschinensprache-Programm enthaelt. Diese Datei wird vom Compiler byteweise gelesen und ohne weitere Interpretation an der Stelle des $INLINE-Befehls in den Objektcode aufgenommen.

Dieser Befehl ermoeglicht das direkte ("inline") Einfuegen von Maschinensprache-Befehlen in ein Programm (im Gegensatz zu Maschinenprogrammen, die mit BLOAD oder POKE in einen eigenen Speicherbereich gebracht werden.)

$INLINE darf nur innerhalb einer als INLINE deklarierten Prozedur verwendet werden (siehe SUB INLINE unter Punkt 4.2.9.).

Die tatsaechliche Adresse im Speicher des mit $INLINE definierten Maschinencodes wird vom Compiler bestimmt und ist abhaengig vom restlichen Programm.
Die folgenden Register duerfen innerhalb des Maschinenprogramms nicht veraendert werden bzw. muessen vor und nach einem Aufruf denselben Wert haben: Stacksegment/Stackpointer (SS/SP), Datensegment (DS) und das Register BP. Eventuell uebergebene Parameter koennen durch Indizieren mit BP vom Maschinenprogramm gelesen und/oder veraendert werden.
Ein Programm kann eine beliebige Anzahl von $INLINE-Befehlen enthalten.

Beispiel:

```
SUB Ton1 INLINE
'enthaelt ein Maschinensprache-Programmm, das
'einen etwas merkwuerdigen Ton erzeugt
       $INLINE &HBA, &H00, &H07, &HE4, &H61, &H24
       $INLINE &HFC, &H34, &H02, &HE6, &H61, &HB9
       $INLINE &H40, &H01, &HE2, &HFE, &H4A, &H74
       $INLINE &H02, &HEB, &HF2
END SUB

'******* Hauptprogramm *******

CALL Ton1
END
```

Anlage A

Liste der Kommandos, Anweisungen und Funktionen in alphabetischer Reihenfolge.

Syntax	Wirkungsweise	Seite
v = ABS(x)	Uebergibt den absoluten Wert des Ausdrucks x.	140
v = ASC(x$)	Uebergibt den ASII-Code fuer das erste Zeichen der Zeichenkette x$.	148
v = ATN(x)	Uebergibt den Arcustangens von x.	147
BEEP [Anzahl]	Der Lautsprecher erzeugt einen Ton.	80
v$ = BIN$(n)	Uebergibt den Wert des numerischen Arguments als eine "binaere Zeichenkette", der aus dem Zeichen "0" und "1" besteht.	150
BLOAD Dateiname[,Relativzeiger]	Der Inhalt einer mit BSAVE erzeugten Datei wird in den Speicher geladen.	190
BSAVE Dateiname, Relativzeiger, Laenge	Der Inhalt eines angegebenen Speicherbereiches wird als Datei auf die Diskette kopiert.	189
CALL Prozedurname [(Parameterliste)]	Es wird eine benutzerdefinierte Prozedur aufgerufen.	197
CALL ABSOLUTE Adresse (Parameterliste)	Es erfolgt ein absoluter Aufruf einer in Assemblersprache geschriebenen Routine.	198
CALL INTERRUPT n	Aufruf eines System-Interrupts.	201
v = CDBL(x)	Wandelt x in eine Zahl doppelter Genauigkeit um.	142
v = CEIL(n)	Uebergibt den naechstgroesseren ganzzahligen Wert oder das Argument selbst.	140

CHAIN Dateiname	Eine weitere mit TBASIC erstellte Objektcode-Datei (Dateityp .TBC oder .EXE) wird in den Speicher geladen und die Programmausfuehrung mit dem geladenem Code fortgesetzt.	130
CHDIR Pfad	Ermoeglicht eine Aenderung des aktuellen Verzeichnisses.	202
v$ = CHR$(n)	Uebergibt ein Zeichen, das dem im Argument angegebenen ASCII-Code entspricht.	149
v = CINT(x)	Wandelt x in eine ganze Zahl um.	141
CIRCLE (x,y),r[,Farbe [,Start,End[,Bild]]]	Zeichnen einer Ellipse (Kreis) auf dem Bildschirm.	178
CLEAR	Der Speicherbereich fuer Variable wird geloescht.	74
v = CLNG(x)	Wandelt x in das Format Langinteger.	142
CLOSE [[#]Dateinummer[,[#] Dateinummer]...]	Schliessen der Ein-/Ausgabe zu einer Datei.	160
CLS	Loeschen des Bildschirms.	173
COLOR [Vordergrund][, [Hintergrund] [,Bildschirmrand]]	Setzt im Textmodus die Farben fuer den Vordergrund, Hintergrund und den Bildschirmrand.	175
nach SCREEN 1: COLOR [Hintergrund] [,[Palette]] nach SCREEN 7..SCREEN 10: COLOR [Vordergrund] [,[Hintergrund]]	Setzt im Grafikmodus die Farbe fuer den Hintergrund und waehlt eine von zwei Paletten aus.	175
COM(n) ON/OFF/STOP	Kontrolle der Reaktion des Programms auf Eingaben ueber die seriellen Ports.	171
$COMn Groesse	Reserviert Speicherplatz fuer den Empfangspuffer eines seriellen Ports.	213

v$ = COMMAND$	Diese Funktion uebergibt eventuelle Zusatzparameter, die beim Start des Programms von der DCP-Kommandoebene mit angegeben wurden.	202
COMMON Variable[,Variable...]	Uebergibt eine oder mehrere Variablen zwischen zwei verketteten Programmen.	131
v = COS(x)	Uebergibt den Cosinus von x.	146
v = CSNG(x)	Wandelt x in eine Zahl einfacher Genauigkeit um.	141
v = CSRLIN	Uebergeben der vertikalen Koordinate (Zeilenposition) des Kursors.	175
V% = CVI(2-Byte-Zeichenkette) V& = CVL(4-Byte-Zeichenkette) V! = CVS(4-Byte-Zeichenkette) V# = CVD(8-Byte-Zeichenkette)	Umwandeln von Daten aus einer Direktzugriffsdatei in die numerische Form.	165
y! = CVMS(4-Byte-Zeichenkette) y# = CVMD(8-Byte-Zeichenkette)	Konvertieren der Inhalte von Direktzugriffsdateien, die im Format des BASIC-Interpreters BASI aufgezeichnet wurden, in numerischer Form.	166
DATA Konstante [,Konstante...]	Definieren von Konstanten fuer READ-Anweisungen.	88
DATE$ = x$ als Anweisung v$ = DATE$ als Variable	Setzt und liest das computerinterne Datum.	78
DECR Variablenname [,Wert]	Verringert eine Variable.	82
- Einzeilige Funktion: DEF FNBezeichner[(Argumentenliste)] = Ausdruck - Mehrzeilige Funktion: DEF FNBezeichner[(Argumentenliste)] [LOCAL Variablenliste] [STATIC Variablenliste] [SHARED Variablenliste] . . Anweisungen . [EXIT DEF] [FNBezeichner = Ausdruck] END DEF	Definiert eine einzeilige (DEF FN) oder mehrzeilige Funktion (DEF FN/END DEF).	115

DEF SEG [=Segment]	Legt fest, auf welches Speichersegment sich nachfolgende Adressangaben fuer BLOAD, BSAVE, CALL ABSOLUTE, PEEK und POKE beziehen.	188
DEFTyp [Buchstabenbereich [,Buchstabenbereich]...	Definiert Variablentypen ueber das erste Zeichen ihres Namens.	80
DELAY Sekunden	Haelt die Programmausfuehrung fuer einen definierten Zeitraum an.	90
DIM STATIC|DYNAMIC Bezeichner(Index [,Index]...)[,Bezeichner(Index[,Index]...)]... DIM STATIC|DYNAMIC Bezeichner(Bereich [,Bereich]...)[,Bezeichner(Bereich[,Bereich]...)]	Festlegen und Erzeugen von Feldern.	109
DO [WHILE/UNTIL Ausdruck] . . Anweisungen . [EXIT LOOP] . LOOP [WEND] [WHILE/UNTIL Ausdruck]	Zyklische Abarbeitung einer Schleife mit Test auf TRUE/FALSE am Beginn und/oder am Ende der Schleife.	102
DRAW Zeichenkette	Zeichnen eines grafischen Objektes auf dem Bildschirm.	179
$DYNAMIC	Setzt die Standardvorgabe des Compilers zum Erzeugen von Feldern auf "dynamisch".	210
END [DEF|IF|SELECT|SUB]	Beendet die Strukturdefinition oder die Programmausfuehrung.	90
v = ENDMEM	Uebergibt die Adresse der hoechsten zur Verfuegung stehenden Speicherzelle.	195
ENVIRON Zeichenkettenketten-Ausdruck	Aendert Parameter in der BASIC-Umgebungstabelle.	204
v$ = ENVIRON$ (Parametername) v$ = ENVIRON (n)	Uebergibt die eingetragenen Parameter der BASIC-Umgebungstabelle.	205

v = EOF(Dateinummer)	Zeigt an, ob das Ende der Datei erreicht wurde.	168
v = ERADR	Uebergibt die Adresse des zuletzt aufgetretenen Fehlers.	135
ERASE Feldname [,Feld-name...]	Loeschen dynamischer Felder bzw. Zuruecksetzen statischer Felder.	113
v = ERDEV v$ = ERDEV$	Es wird der Fehlerstatus eines Peripheriegeraetes bzw. des dazugehoerigen Treiber-programms ermittelt.	206
v = ERR v = ERL	Bestimmt den Fehlercode und die Zeilennummer, in der der Laufzeitfehler aufgetreten ist.	133
ERROR n	Gezieltes Erzeugen eines Laufzeitfehlers.	133
$EVENT ON/OFF	Kontrolliert das Erzeugen zusaetzlicher Codes fuer das Ueberwachen von Ports.	214
EXIT SELECT|DEF|FOR|IF|LOOP|SUB	Verlaesst eine Befehlsstruk-tur vor dem eigentlichen Ende, d.h. es werden eine oder mehrere Anweisungen uebersprungen.	106
v = EXP(x)	Errechnet e hoch x.	143
v = EXP2(x)	Errechnet 2 hoch x.	143
v = EXP10(x)	Errechnet 10 hoch x.	144
FIELD [#]Dateinummer, Laenge AS Zeichen-kettenvariable [,Laenge AS Zeichenketten-variable]...	Definition von Feldvariablen als Puffer einer Direktzu-griffsdatei.	161
FILES [Dateiname]	Anzeige des Bibliotheksver-zeichnisses (Directory).	77
v = FIX(x)	Konvertiert x in einen ganz-zahligen Wert durch Abschnei-den der Nachkommastelle.	140

FOR Variablenname = xTOy . [STEP z] . . Anweisungen . NEXT [Variablenname [,Variablenname...]	Fuehrt einen Zyklus von Anweisungen aus, wobei die Anzahl der Durchlaeufe ueber eine Laufvariable festgelegt wird, die bei jedem Durchlauf automatisch erhoeht bzw. verringert wird.	97
v = FRE(Zeichenketten-Ausdruck|-1|-2)	Es wird der freie Speicherplatz ermittelt.	156
GET (x1,y1) - (x2,y2), Bereich	Lesen von Punkten aus einem Bereich des Bildschirms in ein numerisches Feld.	179
GET [#]Dateinummer [,Satznummer]	Lesen eines Datensatzes aus einer Direktzugriffsdatei in einen Puffer fuer Direktzugriff.	162
GET$ [#]Dateinummer, Anzahl, Zeichenkettenvariable	Lesen einer Zeichenkette von einer Binaerdatei.	164
GOSUB Bezeichner	Aufruf eines BASIC-Unterprogramms.	93
GOTO Bezeichner	Es wird ein unbedingter Sprung ausgefuehrt.	91
v$ = HEX$(n)	Uebergibt eine Zeichenkette, die den hexadezimalen Wert eines dezimalen Arguments darstellt.	150
IF Integer-Ausdruck[,] THEN Anweisung(en) [ELSE Anweisung(en)]	Pruefen einer Bedingung und Aendern des Programmablaufs in Abhaengigkeit von der Bedingung.	99
IF Integer-Ausdruck[,] . THEN . Anweisung(en) . [ELSEIF Integer-Ausdruck . [,] THEN . Anweisung(en) [ELSE . Anweisung(en)] . END IF	Pruefen einer Serie von Vergleichen als Blockstruktur und Aendern des Programmablaufs in Abhaenigkeit von der Bedingung.	100

$IF Konstante . . Anweisung(en) . [$ELSE . . Anweisung(en) .] $ENDIF	Bedingtes Compilieren.	208
INCR Variablenname [,Wert]	Erhoeht eine Variable.	81
$INCLUDE Dateinamen-Konstanten	Einfuegen einer zusaetzlichen Quelltext-Datei waehrend des Uebersetzens.	211
v$ = INKEY$	Lesen der Tastatur ohne gleichzeitige Anzeige der gelesenen Zeichen auf dem Bildschirm (Echo).	153
$INLINE Bytefolge| Dateinamen-Konstante	Direktes Einfuegen von Maschinensprach-Befehlen in den Programmcode.	220
v = INP(n)	Uebergibt ein gelesenes Byte vom Port n.	194
INPUT [;]["Bedienerfuehrung"|,] Variablenliste	Eingabe eines oder mehrerer Werte ueber die Tastatur waehrend der Programmausfuehrung und Zuweisen an bestimmte Variablen.	82
v$ = INPUT$(n[,[#]Dateinummer])	Liest eine bestimmte Anzahl Zeichen von der Tastatur oder aus einer Datei.	154
INPUT #Dateinummer, Variablenliste	Einlesen von Daten aus einer sequentiellen Datei.	161
v = INSTAT	Uebergibt den momentanen Status der Tastatur bzw. des Tastaturpuffers.	152
v = INSTR([n,]Zielzeichenkette,Suchzeichenkette)	Durchsucht eine Zeichenkette nach einer Zeichenfolge und uebergibt ihre Position innerhalb der Zeichenkette.	148
v = INT(x)	Uebergibt den ganzzahligen Wert des Ausdrucks x.	140

IOCTL [#]Dateinummer, Zeichenketten-Ausdruck v$ = IOCTL$([#]Dateinummer)	Kontrolle und Kommunikation mit dem Treiberprogramm des Peripheriegeraetes.	206
KEY(n) ON/OFF/STOP	Aktiviert bzw. inaktiviert die Unterbrechung ("Abfangen") einer zuvor definierten Tasten (-Kombination).	136
KEY ON/OFF/LIST KEY n Zeichenkette KEY n, CHR$(Tastenzeichen) +CHR$(Suchcode)	Setzt definierte Funktionstasten, sowie Anzeige der Belegung auf dem Bildschirm.	84
KILL Dateiname	Eine Diskettendatei wird geloescht.	77
LBOUND (Feldname[,Dimension])	Uebergibt die kleinstmoegliche Indexnummer eines Feldes bzw. einer Felddimension.	157
x$ = LCASE$(y$)	Verwandelt Grossbuchstaben der Zeichenkette in Kleinbuchstaben.	155
v$ = LEFT$(x$, n)	Uebergibt den linken Teil einer Zeichenkette mit der Laenge n.	151
v = LEN(x$)	Uebergibt die Laenge der Zeichenkette x$.	149
[LET] Variablenname = Ausdruck	Ordnet den Wert eines Ausdrucks einer Variablen zu.	81
LINE [(x1,y1)]-(x2,y2)[,[Farbe][,[B[F]][,Stil]]]	Zeichnen einer Linie oder eines Vierecks auf dem Bildschirm.	178
LINE INPUT [;]["Bedienerfuehrung";] Zeichenkettenvariable	Eingabe einer Zeile von der Tastatur in eine Zeichenketten-Variable.	83
LINE INPUT #Dateinummer, Zeichenkettenvariable	Einlesen einer Zeile von einer sequentiellen Datei in eine Zeichenkettenvariable. Trennzeichen werden ignoriert.	161
V = LOC(Dateinummer)	Uebergibt die aktuelle Position in der Datei.	167

`LOCAL Variablenliste`	Deklaration von Variablen als "lokal" innerhalb einer Prozedur- oder Funktionsdefinition.	125
`LOCATE [Zeile][,[Spalte][,[Anzeige][,[Start][,Stop]]]]`	Setzen des Kursors auf dem Bildschirm und Festlegen der Groesse des Kursors.	174
`V = LOF(Dateinummer)`	Uebergibt die Groesse einer Datei.	168
`v = LOG(x)`	Uebergibt den natuerlichen Logarithmus (Basis e) von x.	144
`v = LOG2(x)`	Uebergibt den Logarithmus von x zur Basis 2.	144
`v = LOG10(x)`	Uebergibt den Logarithmus von x zur Basis 10.	145
`v = LPOS(n)`	Ermittelt die momentane Druckposition im Druckpuffer.	158
`LPRINT [Liste von Ausdruecken][;]` `LPRINT USING Formatzeichenkette; Liste von Ausdruecken[;]`	Ausgabe von Zeichen auf dem Drucker (LPT1:).	84
`LSET Zeichenkettenvariable = Zeichenkettenausdruck` `RSET Zeichenkettenvariable = Zeichenkettenausdruck`	Uebergeben von Daten in den Puffer einer Direktzugriffsdatei.	162
`MEMSET Adresse`	Legt die obere Grenze des von TBASIC benutzten Speicherbereichs fest.	196
`v$ = MID$(x$, n[,m])` als Funktion `MID$(v$, n[,m]) = y$` als Anweisung	Uebergibt eine Anzahl von Zeichen aus der Mitte einer Zeichenkette.	151
`MKDIR Pfad`	Erstellt ein Verzeichnis (Subdirectory) auf der Diskette.	202

`V$ = MKI$(Integer-Ausdruck)`
`V$ = MKL$(Langinteger-Ausdruck)`
`V$ = MKS$(Real-Ausdruck einfacher Genauigkeit)`
`V$ = MKD$(Real-Ausdruck doppelter Genauigkeit)`
	Umwandeln von numerischen Werten in Zeichenketten fuer das Schreiben in eine Direktzugriffsdatei.	167

V$ = MKMS$(Realausdruck einfacher Genauigkeit) V$ = MKMD$(Realausdruck doppelter Genauigkeit)	Konvertieren von Realwerten in Zeichenketten fuer das Schreiben von Direktzugriffs-dateien im Format, gefordert von BASI.	167
MTIMER (Anweisung zum Setzen) v = MTIMER (Funktionen zum Lesen)	Lesen und Setzen des Mikro-sekundenzaehlers.	79
NAME Dateinamel AS Datei-name2	Eine Datei erhaelt einen neuen Namen.	76
NEXT[Variable[,Variable]..]	Schliesst eine FOR...NEXT-Schleife ab.	97
v$ = OCT$(n)	Uebergibt eine Zeichenkette, der den oktalen Wert eines dezimalen Arguments dar-stellt.	150
ON COM(n) GOSUB Label	Festlegen eines Unterpro-gramms zur Behandlung von Zeichen, die ueber die seriellen Ports empfangen werden.	171
ON ERROR GOTO Label	Festlegen einer Routine zur Behandlung von Laufzeit-fehlern.	134
ON n GOTO Label [,Label...] ON n GOSUB Label [,Label..]	Ruft abhaengig vom Wert eines numerischen Ausdrucks eines von mehreren Programmteilen bzw. eines von mehreren Unter-programmen auf.	94
ON KEY(n) GOSUB Label	Festlegen einer "Abfang-routine" (Unterbrechung) fuer eine bestimmte Tasten-kombination.	137
ON PLAY(n) GOSUB Label	Spielen von Musik waehrend der Programmunterbrechung.	187
ON TIMER(n) GOSUB Label	Legt ein Unterprogramm fest, das in regelmaessigen Ab-staenden aufgerufen werden soll.	138

OPEN Dateiangabe [FOR Modus] AS [#]Dateinummer [LEN = Satzlaenge] oder OPEN Modus2, [#]Dateinummer, Dateiangabe [,Satzlaenge]	Eroeffnen einer Datei fuer nachfolgende Lese- und Schreibaktionen.	159
OPEN "COMn:[Geschwindigkeit][,Paritaet] [,Daten][,Stop][,Optionen]" AS [#]Dateinummer [LEN= Groesse]	Eroeffnet eine Datei fuer Datenfernverarbeitung.	169
OPTION BASE Integer-Ausdruck	Es wird eine Untergrenze fuer das Indizieren von Feldern festgelegt.	112
OUT n,m	Sendet ein Byte (8Byte) zu einem I/O-Port.	194
PAINT (x,y)[,[Farbe], [Rand],[Hintergrund]	Fuellen eines Bildschirmschirmbereiches mit Farbe.	180
PALETTE (Attributwert,Farbe) PALETTE USING FELD (Index)	Zuordnung von Attribut-Werten und physikalischen Farben ueber eine Farb-Palette (nur EGA-Karten).	183
v = PEEK (Adresse)	Uebergibt ein gelesenes Byte von der angegebenen Speicherzelle.	189
PLAY Zeichenkette	Erzeugen von Toenen mit Hilfe einer Zeichenkette.	186
v = PLAY(n)	Uebergeben der Anzahl Noten, die sich noch im Puffer fuer Hintergrundmusik befinden.	188
PLAY ON PLAY OFF PLAY STOP	Aktivieren/Inaktivieren der Programmunterbrechung.	187
v = PMAP(x,n)	Umrechnen der physikalischen Koordinaten des Bildschirms in globale Koordinaten und umgekehrt.	185
POKE Adresse, Wert	Schreibt ein Byte in eine Speicherzelle.	189
v = POS(n)	Ermittelt die momentane Spaltenposition des Kursors.	158 175

v = POINT (x,y) v = POINT (n)	Uebergeben des Attribut-Wertes eines Punktes oder der Position des Kursors auf dem Grafikbildschirm.	186
PRINT [Liste von Aus-druecken][;|] [Liste von Aus-druecken][;|]	Ausgabe von Daten auf dem Bildschirm.	83
PRINT #Dateinummer, [USING V$;] Liste von Aus-druecken [;]	Schreiben von Daten in eine sequentielle Datei.	160
PRINT SPC(n)	Uebergibt n Leerzeichen.	158
PRINT USING Formatzeichen-kette; Liste von Ausdruecken[;]	Formatierte Ausgabe von Daten auf dem Bildschirm.	83
PSET (x,y)[,Farbe] PRESET (x,y)[,Farbe]	Zeichnen eines Punktes auf dem Bildschirm.	178
PUT (x,y), Bereich[Aktion]	Faerben eines Bereichs des Bildschirmes.	180
PUT [#]Dateinummer [,Satz-nummer]	Schreiben eines Datensatzes aus dem Puffer fuer Direktzu-griff in eine Direktzugriffs-datei.	162
PUT$ [#]Dateinummer, Zeichenkettenausdruck	Schreiben von Daten in eine Binaerdatei.	163
RANDOMIZE [n] RANDOMIZE TIMER/MTIMER	Setzt einen Startwert fuer einen Zufallszahlengenerator.	82
READ Variable[,Variable...]	Liest Elemente von DATA-An-weisungen und weist diese Werte Variablen zu.	88
REG Register, Wert als Anweisung v = REG(Register) als Funktion	Lesen bzw. Setzen der In-halte des Puffers fuer die Prozessor-Register.	190
REM Bemerkung	Einfuegen erklaerender Be-merkungen (Kommentare) in ein Programm.	78
RESET	Saemtliche evtl. noch nicht aufgezeichneten Dateipuffer-inhalte werden aufgezeichnet und danach alle Dateien ge-schlossen.	76

`RESTORE [Label]`	Setzt den DATA-Zeiger des Programms auf das erste oder auf eine bestimmtes Element.	89
`RESUME [0/NEXT/Label]`	Setzt die Programmausfuehrung nach einer Fehlerbehandlung fort.	136
`RETURN [Label]`	Beendet ein Unterprogramm und fuehrt einen Ruecksprung aus.	95
`v$ = RIGHT$(x$, n)`	Uebergibt den rechten Teil einer Zeichenkette mit der Laenge n.	151
`RMDIR Pfad`	Loescht das angegebene Verzeichnis von der Diskette.	202
`v = RND[(x)]`	Uebergibt eine Zufallszahl zwischen 0 und 1.	156
`RUN [Dateiname]`	Start eines Programms.	72
`SCREEN[Modus][,[Aendern] [,Aseite][,Vseite]]]`	Setzen der Bildschirmattribute.	174
`v = SCREEN (Zeile,Spalte) [,z]`	Uebergeben des ASCII-Codes des Zeichens an der angegebenen Position.	186
`SEEK [#]Dateinummer, Position`	Setzen der Position innerhalb einer als BINARY eroeffneten Datei fuer ein nachfolgendes Lesen oder Schreiben.	162
`$SEGMENT`	Daklaration eines neuen Code-Segments.	217

`SELECT CASE Ausdruck`
`  CASE Pruefungen`
`  .`
`  .  Anweisung(en)`
`  .`
`  [CASE Pruefungen`
`  .`
`  .  Anweisung(en)`
`  .  ]...`
`  [CASE ELSE`
`  .`
`  .  Anweisung(en)`
`  .  ]`
`END SELECT`
	Bilden einer Blockstruktur fuer Pruefungen mit mehreren Moeglichkeiten.	104

v = SGN(x)	Ermittelt das Vorzeichen von x.	141
SHARED Variablenliste	Deklaration von Variablen innerhalb einer Prozedur- oder Funktionsdefinition als "global".	129
SHELL [Befehlszeichen-kette]	Von TBASIC aus wird ein anderes Programm geladen und ausgefuehrt. Anschliessend wird zu TBASIC zurueckverzweigt und das urspruengliche BASIC-Programm fortgesetzt.	203
v = SIN(x)	Errechnet den Sinus von x.	146
SOUND Freq,Dauer	Erzeugen eines Tones ueber den Lautsprecher.	186
$SOUND Puffergroesse	Legt die Groesse des Puffers fuer PLAY fest.	212
v$ = SPACE$(n)	Uebergibt eine Zeichenkette, die aus n Leerzeichen besteht.	152
v = SQR(x)	Uebergibt die Quadratwurzel von x.	145
$STACK Groesse	Legt die Groesse des Stack fest, der dem Programm zur Verfuegung steht.	216
STATIC Variablenliste	Deklaration von Variablen innerhalb einer Funktions- oder Prozedurdefinition als statisch.	127
$STATIC	Setzt die Standardvorgabe des Compilers zum Erzeugen von Feldern auf "statisch".	209
STOP	Bricht die Programmausfuehrung ab.	89
v$ = STR$(x)	Wandelt den Wert eines numerischen Ausdrucks in eine Zeichenkette um.	151
v$ = STRING$(n, m) oder v$ = STRING$(n, x$)	Uebergibt eine Zeichenkette der Laenge n, deren Zeichen alle aus dem ASCII-Code m oder dem ersten Zeichen von x$ bestehen.	152

SUB Bezeichner[(Parameter- liste)] [INLINE] [LOCAL Variablenliste] [STATIC Variablenliste] [SHARED Variablenliste] . . Anweisung(en) . [EXIT SUB] END SUB	Definieren einer Prozedur (Subprogramme).	118
SWAP Variable 1, Variable 2	Vertauscht die Werte zweier Variablen.	81
SYSTEM	Ein Programm wird beendet.	76
PRINT TAB(n)	Setzt den Kursor auf die angegebene Position n.	158
v = TAN(x)	Errechnet den Tangens von x.	146
TIME$ = x$ (Setzen der Uhrzeit) v$ = TIME$ (Lesen der Uhrzeit)	Enthaelt die Uhrzeit des Systems, die sowohl gesetzt oder gelesen werden kann.	78
v = TIMER	Ermittelt die Anzahl der seit 0 Uhr 00 bzw. seit dem Start des Computers vergangenen Sekunden.	154
TRON TROFF	Ein schrittweises Verfolgen des Programmablaufs zur Fehlersuche ist damit moeglich.	74
v = UBOUND(Feldname[, Dimension])	Ermittelt die Nummer des hoechsten Elements eines Feldes bzw. einer Felddimension.	157
x$ = UCASE$(y$)	Verwandelt Kleinbuchstaben der Zeichenkette in Grossbuchstaben.	155
v = VAL(x$)	Uebergibt den numerischen Wert der Zeichenkette x$.	149
v = VARPTR(Variable)	Uebergibt die Adresse einer Variablen.	191
v$ = VARPTR$(Variable)	Uebergibt den Zeiger zu einer Variablen in Form einer Zeichenkette.	192

v = VARSEG(Variable)	Uebergibt die Segmentadresse einer Variablen.	192
VIEW [[SCREEN][(x1,y1)-(x2,y2) [,[Farbe][,Rand]]]]	Definieren eines rechteckigen Bereiches auf dem Bildschirm.	185
WAIT Port, Bitmuster, Maske	Wartet auf einen bestimmten Wert, der von einem ausgewaehlten Port gelesen wird.	194
WIDTH [Geraetename,| Dateinummer,] Breite	Legt die Breite einer Ausgabezeile des Bildschirms, eines Peripheriegeraetes oder einer Datei fest.	87
WINDOW [[SCREEN] (x1,y1)-(x2,y2)]	Neudefinieren der Bildschirmkoordinaten.	185
WHILE Integer-Ausdruck . . Anweisung(en) . WEND	Fuehrt einen Zyklus von Anweisungenaus, wobei die Anzahl der Durchlaeufe variabel ist, abhaengig von einer angegebenen Bedingung.	101
WRITE [Liste von Ausdruecken]	Schreibt Daten auf den Bildschirm, getrennt durch Kommas.	84
WRITE #Dateinummer, Liste der Ausdruecke	Schreiben von Daten in eine sequentielle Datei.	161

Anlage B

Reservierte Woerter

Einige Woerter bei TBASIC haben eine bestimmte Bedeutung. Diese Woerter nennt man **Reservierte Woerter.**

Reservierte Woerter beinhalten alle TBASIC-Kommandos, -Anweisungen, -Funktionsnamen und -Operatornamen.
Reservierte Woerter duerfen nicht als Variablennamen benutzt werden.
Reservierte Woerter muessen immer von Daten oder anderen Teilen der TBASIC-Anweisung durch Leerstellen oder andere Sonderzeichen, die in der Syntax erlaubt sind, getrennt werden. Das bedeutet, dass reservierte Woerter immer getrennt stehen muessen, so dass sie von TBASIC erkannt werden koennen.

Die folgende Liste zeigt alle reservierten Woerter:

ABS	CVS	FIELD
ABSOLUTE	DATA	FILES
AND	DATE$	FIX
ASC	DECR	FN
ATN	DEF	FOR
BEEP	DEFDBL	FRE
BIN$	DEFINT	GET
BLOAD	DEFLNG	GET$
BSAVE	DEFSNG	GOSUB
CALL	DEFSTR	GOTO
CALL ABSOLUTE	DELAY	HEX$
CALL INTERRUPT	DIM	IF
CDBL	DO	$IF
CEIL	DRAW	IMP
CHAIN	$DYNAMIC	$INCLDE
CHDIR	ELSE	INCR
CHR$	$ELSE	INKEY$
CINT	END	INLINE
CIRCLE	$ENDIF	INP
CLEAR	ENDMEN	INPUT
CLOSE	ENVIRON	INPUT$
CLNG	ENVIRON$	INPUT#
CLS	EOF	INSTAT
COLOR	EQF	INSTR
COM	ERADR	INT
$COM	ERASE	INTERRUPT
COMMON	ERDEV	IOCTL
COS	ERDEV$	IOCTL$
CSNG	ERL	KEY
CSRLIN	ERR	KILL
CVD	ERROR	LCASE$
CVI	EXIT	LEFT$
CVL	EXP	LEN
CVMD	EXP2	LET
CVMS	EXP10	LINE

```
LINE INPUT
LINE INPUT#
LOC
LOCAL
LOCATE
LOF
LOG
LOG2
LOG10
LOOP
LPOS
LPRINT
LSET
MEMSET
MID$
MKDIR
MKD$
MKI$
MKL$
MKMD$
MKMS$
MKS$
MOD
NAME
NEXT
NOT
OCT$
OFF
ON
OPEN
OPTION
OPTION BASE
OR
OUT
PAINT
PALETTE
PEEK
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PRINT#
PSET
PUT
PUT$
RANDOMIZE
READ
REG
REM
RESET
RESTORE
RESUME
RETURN
RIGHT$
RMDIR
RND
RSET
RUN
SCREEN
SEEK
SEG
$SEGMENT
SELECT
$SEVENT
SGN
SHARED
SHELL
SIN
SOUND
$SOUND
SPACE$
SPC
SQR
$STACK
STATIC
$STATIC
STEP
STOP
STR$
STRING$
SUB
SWAP
SYSTEM
TAB
TAN
THEN
TIME$
TIMER
TO
TROFF
TRON
UCASE$
USING
USING#
VAL
VARPTR
VARPTR$
VARSEG
VIEW
WAIT
WEND
WHILE
WIDTH
WINDOW
WRITE
WRITE#
XOR
```

Anlage C

Fehlercodes und Fehlermeldungen

TBASIC kennt zwei Arten von Fehlern:

- **Fehler waehrend des Compilierens:**
  Dies sind syntaktische Fehler, die der Compiler entdeckt.

- **Laufzeitfehler:**
  Sie werden nicht vom Compiler selbst, sondern von Pruefroutinen erkannt, die der Compiler zusammen mit dem Objektcode erzeugt hat.

Der ueberwiegende Teil der **Compiler-Fehlermeldungen** laesst sich auf vergessene oder ueberzaehlige Symbole, falschgeschriebene Kommandos, nicht korrekt abgeschlossene Klammerebenen usw. zurueckfuehren. Findet der Compiler im Quelltext etwas, was er nicht versteht (oder nicht verstehen will, weil es unzulaessig ist), dann wird automatisch der Editor aufgerufen. In der Statuszeile des Editors erscheint die Fehlermeldung im Klartext, der Kursor befindet sich auf der Stelle, an der der Compiler den Fehler bemerkt hat. Durch den Compiler entdeckte Fehler haben Nummern von 256 und darueber.

Der erste Tastendruck beseitigt die Fehlermeldung. Nach Korrektur des Fehlers wird der Editor verlassen und erneut compiliert.

Laufzeitfehler koennen nur nach dem Start eines compilierten Programms auftreten. Beispiele fuer diese Art von Fehlern sind Probleme bei der Diskettenarbeit (Diskette voll oder schreibgeschuetzt, nicht gefundene Dateien oder Directories), illegale Funktionsaufrufe (wie z.B. **COLOR** mit dem Argument 713 fuer die Zeichenfarbe) oder z.B. Probleme mit dem Speicherplatz.

Laufzeitfehler koennen (im Gegensatz zu Fehlermeldungen des Compilers) programmgesteuert abgefangen werden. Mit der Anweisung **ON ERROR GOTO** kann bestimmt werden, dass im Falle eines Fehlers nicht mit einer Meldung abgebrochen, sondern zu einem bestimmten Programmteil gesprungen wird. Der durch **ON ERROR GOTO** festgelegte Programmteil kann die Stelle und Art des Fehlers bestimmen und abhaengig davon Korrekturmassnahmen ausfuehren bzw. Befehle wiederholen. Fehler bei Operationen mit Peripheriegeraeten (speziell: Diskettenlaufwerken) koennen auf diese Weise problemlos beseitigt werden.

Mit dem Befehl **ERROR** lassen sich Laufzeitfehler beliebiger Art zum Pruefen von Fehlerbehandlungsroutinen erzeugen. Die jeweiligen Beschreibungen von **ON ERROR GOTO**, **ERR**, **ERL** und **ERADR** in Kapitel 4 enthalten weitere Details zum Thema "Fehlerbehandlung".

Wenn innerhalb eines Programms keine eigene Fehlerbehandlung vorgesehen ist (d.h. die Anweisung **ON ERROR GOTO** wurde nicht programmiert), dann wird die Ausfuehrung im Fehlerfall abgebrochen. Wenn das Programm von der DCP-Kommandoebene aus gestartet wurde, wird eine Fehlernummer und der Stand des Programmzaehlers zum Zeitpunkt des Fehlers angezeigt. Wird das Programm innerhalb der Programmierumgebung von TBASIC gestartet, dann kommt noch eine Bezeichnung des Fehlers im Klartext zu diesen beiden Zahlen hinzu.

Laufzeitfehler haben Nummern im Bereich von 1 bis 255. Eine Fehlermeldung sieht wie folgt aus:

Fehler 5 Illegaler Funktionsaufruf, pgm-ctr: 761

Ueber die Option **Run-time error** im Menue **Debug** kann durch Eingabe des Programmzaehler-Standes die entsprechende Stelle im Quelltext lokalisiert werden (siehe Kapitel 2, Abschnitt "Das Menue **Debug**").

Als ".EXE" oder ".TBC" compilierte Dateien enthalten keine Fehlermeldungen im Klartext. Dadurch werden pro Programm einige Kilobyte an Platz auf der Diskette eingespart. Im Falle eines Fehlers wird aber auch nur eine entsprechende Nummer ausgegeben.

Zusammenstellung der Fehlermeldungen:

Laufzeitfehler haben Nummern im Bereich von 1 bis 255. Syntaxfehler entfallen, da diese bereits vom Compiler entdeckt werden.

1 **NEXT ohne FOR**
Der naechste auszufuehrende Programmschritt ist ein **NEXT**, allerdings wurde vorher keine entsprechende **FOR**-Anweisung ausgefuehrt. In den meisten Faellen bedeutet das, dass mit **GOTO** in eine **FOR**-Schleife hineingesprungen wurde - Fehler wie ueberkreuzende **FOR/NEXT**-Schleifen oder falschgeschriebene Variablen bei **NEXT** werden bereits vom Compiler entdeckt.

2 **Falscher Datentyp bei READ**
Das Programm hat versucht, mit dem Befehl **READ** eine Zeichenkette in eine numerische Variable einzulesen.

3 **RETURN ohne GOSUB**
Der naechste auszufuehrende Programmschritt ist ein **RETURN**, allerdings wurde vorher kein entsprechendes **GOSUB** ausgefuehrt. Dieser Fehler kann entweder durch ein versehentliches Hineinspringen oder durch das "Hineinfallen" des Programms in ein Unterprogramm ausgeloest werden. Im letzteren Fall sollte geprueft werden, ob das Hauptprogramm auch wirklich mit einem **END**-Befehl endet.
Achtung: Diese Fehlermeldung erhaelt man nur, wenn der Schalter **Stack test** (Menue **Options**) auf **ON** gesetzt ist !

4 **Keine weiteren DATA-Elemente**
("Out of Data") Das Programm hat versucht, einen **READ**-Befehl auszufuehren, obwohl bereits saemtliche **DATA**-Elemente gelesen worden sind.

5 **Illegaler Funktionsaufruf**
Das ist eine Fehlermeldung fuer alle Faelle, in denen keine genauere Angabe moeglich ist. Sie wird durch die Uebergabe eines unzulaessigen Arguments oder durch den unzulaessigen Aufruf einer Funktion ausgeloest. Dies koennte z.B. sein:
- Anwendung von **COLOR** auf einen Monochrom-Adapter (SCREEN);
- Angabe einer unzulaessigen Groesse wie **COLOR 3265**;
- mathematische Unmoeglichkeiten wie negative Wurzeln oder Logarithmen;
- **ASC** mit einer Null-Zeichenkette als Argument.

6 **Ueberlauf**
Das Ergebnis einer Rechenoperation uebersteigt den zulaessigen Wertebereich des verwendeten Formats. Ueberlaeufe bei Operationen mit Realzahlen werden immer entdeckt, bei Integer-Berechnungen dagegen nur dann, wenn der Schalter **Overflow** im Menue **Options** vor dem Compilieren auf **ON** gesetzt ist.

7 **Nicht genug Speicherplatz**
("Out of Memory") Diese Meldung kann von allen Befehlen erzeugt werden, die zusaetzlichen Speicherplatz des Computers belegen (dynamische Dimensionierungen innerhalb des Hauptprogramms usw.). In diese Kategorie faellt auch der Versuch, die Obergrenze des Speichers mit **MEMSET** zu weit herabzusetzen.
Die Fehler "Zeichenkettenspeicherplatz voll" und "Nicht genug Speicherplatz zum Start" haben jeweils eigene Nummern.

8 nicht definiert

9 **Index unzulaessig**
Das Programm hat versucht, auf ein Feld-Element zuzugreifen, das ausserhalb der zuvor mit **DIM** gesetzten Indexgrenzen liegt. Diese Fehlermeldung wird nur dann erzielt, wenn vor dem Compilieren der Schalter **Bounds** im Menue **Options** auf **ON** gesetzt wurde. In diese Kategorie fallen auch Zugriffe auf zuvor nicht dimensionierte Felder mit einem Index groesser 10.

10 **Doppelte Definition**
Das Programm hat versucht, ein Feld mehrfach zu dimensionieren (d.h. entweder in einer Schleife ein Feld ein zweites Mal definiert oder nach dem Loeschen eines statischen Feldes mit **ERASE** ein erneutes Dimensionieren vorzunehmen).

11 **Division durch Null**
Das Programm hat eine Division durch Null ausgefuehrt, entweder in der Form **numerischer Ausdruck/0** oder **0^x**, wobei **x** einen negativen Wert darstellt. Dieser Fehler bewirkt lediglich die Ausgabe einer Fehlermeldung - das Programm arbeitet danach mit dem groesstmoeglichen Wert weiter, der im momentan verwendeten Format darstellbar ist. Mit **ON ERROR GOTO** kann eine Division durch Null nicht abgefangen werden.

12 nicht definiert

13 nicht definiert

14 **Zeichenkettenspeicherplatz voll**
Fuer Zeichenketten stehen maximal 64 KByte Speicherplatz zur Verfuegung.

15 **Zeichenkettenlaenge groesser 32767**
Das (Zwischen-)Ergebnis einer Zeichenkettenoperation hat mehr als 32767 Zeichen.

16 bis 18 nicht definiert

19 **ON ERROR-Routine ohne END**
Eine mit **ON ERROR GOTO** aufgerufene Routine muss entweder mit **RESUME** oder **END** enden.

20 **RESUME ohne vorhergehenden Fehler**
Der naechste auszufuehrende Programmschritt ist ein **RESUME**, allerdings ist verher ueberhaupt kein Fehler aufgetreten (eine Ruecksprungadresse von der Fehlerbehandlungsroutine aus ist nicht vorhanden).

21 nicht definiert

22 **Fehlender Operand**

23 **Eingabepuffer-Ueberlauf**

24 **Peripherie-Timeout**
Das Programm hat einen seriellen Port als Datei eroeffnet, innerhalb der vorgegebenen Zeitspanne erfolgt keine Reaktion.

25 **Peripherieprobleme**
Das Statuswort eines angesprochenen Peripheriegeraetes (serielle Ports, Druckerschnittstelle, usw.) zeigt eine Fehlerbedingung (elektronischer Natur) an.

26 **FOR ohne NEXT**
Das Programm wurde beendet, waehrend noch eine oder mehrere **FOR**-Schleifen offen waren. Grund: **NEXT**-Anweisungen wurden mit **GOTO** uebersprungen, oder innerhalb einer Schleife steht eine **END**-Anweisung.

27 **Kein Papier im Drucker**
("Out of Paper") Die Schnittstelle eines angesprochenen Druckers meldet, dass kein Druckpapier mehr vorhanden ist. Dieser Fehler kann auch durch ausgeschaltete, ueberhaupt nicht angeschlossene Drucker o.ae. ausgeloest werden.

28 nicht definiert

29 **WHILE ohne WEND**
Das Programm wurde beendet, waehrend noch eine oder mehrere mit **WHILE** gestartete Schleifen offen waren. Grund: Dazugehoerige **WEND**-Anweisungen wurden mit **GOTO** uebersprungen, oder innerhalb einer Schleife steht eine **END**-Anweisung.

30 **WEND ohne WHILE**
Der naechste auszufuehrende Programmschritt ist ein **WEND**, allerdings wurde zuvor keine Schleife mit **WHILE** gestartet. Einzig moeglicher Grund: In die Schleife ist mit **GOTO** hineingesprungen worden.

31 bis 49 nicht definiert

50 **Unpassende FIELD-Groesse**
Mit **FIELD** wurde ein Dateipuffer definiert, dessen Gesamtgroesse die Recordlaenge der dazugehoerigen Datei uebersteigt.

51 **Interner Fehler**
Die Hardware pruefen. Im erneuten Fehlerfalle das Serviceunternehmen ueber die Fehlerbedingungen informieren.

52 **Unzulaessige Dateinummer/nicht OPEN**
Entweder hat das Programm versucht, eine Datei mit einer Nummer zu eroeffnen, die die von DCP gesetzte Grenze (normalerweise 16) ueberschreitet, oder es wurde ein Dateibefehl (**FIELD, READ, WRITE**) fuer eine Datei gegeben, die zum gegenwaertigen Zeitpunkt nicht offen ist.

53 **Datei nicht gefunden**
Eine Datei des angegebenen Namens/Suchwegs existiert nicht. Dieser Fehler wird von **OPEN** nur ausgeloest, wenn die Datei im Modus **INPUT** eroeffnet werden soll - in allen anderen Faellen wird sie automatisch angelegt. Die zweite Moeglichkeit, diesen Fehler zu erzeugen, ist Angabe einer nicht existierenden Datei als "alter Name" fuer **NAME**.

54 **GET($)/PUT($) nicht erlaubt**
Das Programm hat versucht, den Befehl **GET, PUT, GET$** oder **PUT$** auf eine sequentielle Datei anzuwenden.

55 **Datei ist OPEN**
Das Programm hat entweder versucht, eine bereits offene Datei erneut zu eroeffnen oder eine offene Datei mit **KILL** zu loeschen.

56 nicht definiert

57 **Geraete-I/O**
Bei einem Zugriffsversuch auf ein Peripheriegeraet ergab sich eine (momentan) nicht zu behebende Fehlerbedingung.

58 **Dateiname existiert bereits**
NAME kann einer Datei den neuen Namen nicht geben, weil bereits eine Datei dieses Namens (im selben Directory) existiert.

59 nicht definiert

60 nicht definiert

61 **Diskette/Festplatte voll**
Auf dem angegebenen Datentraeger ist nicht genuegend Platz vorhanden.

62 **Dateiende ueberschritten**
Das Programm hat versucht, mehr Daten von einer Datei zu lesen als vorhanden sind. Es ist deshalb vor Leseversuchen ein Pruefen mit der Funktion **EOF** ("End of File" = Dateiende) auszufuehren. Dieser Fehler wird auch erzeugt beim Versuch, eine mit **OUTPUT** oder **APPEND** eroeffnete sequentielle Datei zu lesen.

63 **Illegale Recordnummer**
Das Programm hat versucht, **GET** oder **PUT** zusammen mit einer negativen Recordnummer oder einer Recordnummer > 16 777 215 zu benutzen.

64 **Unzulaessiger Dateiname**
Ein zusammem mit **NAME, FILES** oder **KILL** angegebener Dateiname entspricht nicht den Regeln von DCP.

65 nicht definiert

66 nicht definiert

67 **Root-Directory (Basisverzeichnis) voll**
Subdirectories (Unterverzeichnisse) werden je nach Anzahl der Dateieintraege erweitert, das Root-Directory einer Diskette/Festplatte kann dagegen nur eine begrenzte Anzahl von Eintraegen aufnehmen. Diesen Fehler erhalten Sie unter Umstaenden auch dann, wenn das Programm die **CREATE**-Funktion von DCP durch einen illegalen Dateinamen (zusammen mit **OPEN**) durcheinandergebracht hat.

68 **Peripheriegeraet nicht verfuegbar**
Das Programm hat versucht, ein nicht angeschlossenes Peripheriegeraet anzusprechen bzw. als Geraetedatei zu eroeffnen (z.B. **COM1:** auf einem Computer, der nicht ueber eine entsprechende Zusatzkarte verfuegt).

69 **COM-Puffer: Ueberlauf**
Das Programm hat versucht, in einen bereits vollen Kommunikationspuffer weitere Zeichen einzulesen.

70 **Diskette schreibgeschuetzt**
Das Programm hat versucht, eine Schreiboperation auf einer Diskette auszufuehren, die schreibgeschuetzt ist. Nicht nur **WRITE, KILL, MKDIR** und**RMDIR** stellen eine Schreibaktion dar, sondern auch **NAME** und **OPEN** einer noch nicht existierenden Datei.

71 **Keine Diskette/nicht verriegelt**
("Disk not ready") Auf das angegebene Laufwerk kann nicht zugeriffen werden, weil entweder keine Diskette eingelegt ist oder das Laufwerk nicht verriegelt ist.

72 **Spur/Sektor defekt**
("Disk Media Error") Die Elektronik eines angesprochenen Laufwerks zeigt an, dass ein oder mehrere Sektoren der Diskette/Festplatte schadhaft sind.

73 nicht definiert

74 **NAME kann nicht kopieren**
("Rename across Disks") Das Programm hat versucht, eine Datei von einem Laufwerk in ein anderes "umzubenennen". **NAME** gibt einer Datei lediglich einen neuen Namen, eine Kopie zwischen zwei Laufwerken kann dieser Befehl nicht ausfuehren.

75 **Datei gegen Zugriff gesperrt**
Das Programm hat versucht, ein Subdirectory als Datei zu eroeffnen, eine schreibgeschuetzte Datei/ein Subdirectory mit **KILL** zu loeschen oder ein Subdirectory mit **RMDIR** zu entfernen, das Dateieintraege enthaelt.

76 **Suchweg existiert nicht**
Der bei **CHDIR, MKDIR, RMDIR, OPEN** usw. angegebene Suchweg ist nicht vorhanden. Hauptsaechliche Gruende: Falschgeschriebene Directorynamen oder ein vergessener Schraegstrich zu Anfang eines vollstaendigen Namens. Im letzteren Fall stellt DCP den Namen des momentan gesetzten Standard-Directories voran (und erzeugt unter Umstaenden recht lange Suchwege).

77 bis 202 nicht definiert

203 **Numerik-Coprozessor fehlt !**
Das Programm ist mit der Option **8087** compiliert worden, benoetigt also einen numerischen Coprozessor zur Ausfuehrung (der anscheinend nicht vorhanden ist).

203 **Nicht genuegend Speicherplatz zum Start**
Das Programm selbst konnte zwar noch in den Speicher geladen werden (sonst haette sich DCP mit einem entsprechenden Fehler gemeldet) - bei der Belegung der als statisch definierten Bereiche (Felder, Stack, Zeichenkettenspeicher, PLAY- und Kommunikationspuffer) stellte sich allerdings heraus, dass der Speicherplatz nicht reicht. Die erste moegliche Abhilfe: Compilieren des Programms als **.EXE**-Datei, Beenden von TBASIC und Starten des Programms von der DCP-Kommandoebene aus (**TBASIC.EXE** belegt immerhin rund 200 KByte des Speichers).

204 **CHAIN/RUN innerhalb von TBASIC**
Innerhalb der Programmierumgebung von TBASIC koennen die Befehle **CHAIN** und **RUN Dateiname** nicht ausgefuehrt werden. Compilieren des Programms als **.EXE**-Datei und Starten von der DCP-Kommandoebene aus erforderlich.

211 **SELECT: alle CASEs FALSE**
Innerhalb einer **SELECT**-Anweisung sind alle **CASE**-Pruefungen fehlgeschlagen und ein **CASE ELSE**-Zweig ist nicht angegeben.

242 **Stack-Ueberlauf**
Es wurde mehr Platz auf dem Stack belegt, als zur Verfuegung steht. Dieser Fehler kann eigentlich nur bei rekursiver Programmierung entstehen. Eine "ordentliche" Meldung erhaelt man nur, wenn der Schalter **Stack test** beim Compilieren auf **ON** gesetzt ist - ansonsten stuerzt das System ab.

Anlage D

ASCII-Zeichencodes

In der folgenden Tabelle sind alle ASCII-Codes (dezimal, hexadezimal und ihre zugehoerigen Zeichen) aufgelistet.
Diese Zeichen koennen mit Hilfe von **PRINT CHR$(n)** angezeigt werden, wobei **n** der ASCII-Code ist.
In der Tabelle sind die Standardinterpretationen der ASCII-Codes 0 bis 31 aufgefuehrt. Dies sind nicht darstellbare Zeichen und werden gewoehnlich fuer Steuerfunktionen oder Datenfernverarbeitung benutzt.

Jedes dieser Zeichen kann ueber die Tastatur eingegeben werden, indem man die Taste **"ALT"** betaetigt und niederhaelt und dann die Ziffern fuer den ASCII-Code auf der Zehnertastatur eingibt.

Hinweis:

Die Zeichen fuer die ASCII-Codes 128 bis 255 koennen nur im Textmodus dargestellt werden. Siehe Anweisung **SCREEN** in diesem Handbuch und Abschnitt "Benutzung des Bildschirms" im **BASIC-Handbuch.**

ASCII-Codetabelle

Dez	Hex	CHR	Dez	Hex	CHR	Dez	Hex	CHR	Dez	Hex	CHR
0	00	NUL	32	20		64	40	@	96	60	`
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	EOT	36	24	$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	'	71	47	G	103	67	g
8	08	BS	40	28	(	72	48	H	104	68	h
9	09	HT	41	29	)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

ASCII-Codetabelle (Fortsetzung)

Dez	Hex	CHR	Dez	Hex	CHR	Dez	Hex	CHR	Dez	Hex	CHR
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	∈
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	▒	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	▉	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	n
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	

Erweiterte Codes

Fuer gewisse Tasten und Tastenkombinationen, die nicht im Standard ASCII-Code dargestellt werden koennen, wird ein erweiterter Code von der Systemvariablen **INKEY$** uebergeben. Eine Leerstelle (ASCII-Code 000) wird als erstes Zeichen einer Zwei-Zeichen-Zeichenkette uebergeben. Wird **INKEY$** eine Zeichenkette aus zwei Zeichen empfangen, sollte man zurueckverzweigen und das zweite Zeichen pruefen, um zu bestimmen, welche Taste betaetigt wurde. Im allgemeinen, aber nicht immer, ist dieser zweite Code der Pruefcode der Primaertaste, die betaetigt wurde. Die ASCII-Codes (dezimal) fuer dieses zweite Zeichen und die zugehoerigen Tasten sind unten aufgelistet.

Zweiter Code	Bedeutung
3	(Leerstelle)NUL
15	(Umschalten Tabulator)| <--
16-25	ALT-Q, W, E, R, T, Y, U, I, O, P
30-38	ALT-A, S, D, F, G, H, J, K, L
44-50	ALT-Z, X, C, V, B, N, M
59-68	Funktionstasten F1 bis F10 (falls inaktiviert als Funktionstasten)
71	HOME
72	Kursor nach oben
73	PG UP
75	Kursor nach links
77	Kursor nach rechts
79	END
80	Kursor nach unten
81	PG DN
82	INS
83	DEL
84-93	F11-F20 (Shift-F1 bis F10)
94-103	F21-F30 (CTRL-F1 bis F10)
104-113	F31-F40 (ALT-F1 bis F10)
114	CTRL-PRTSC
115	CTRL-Kursor nach links (vorheriges Wort)
116	CTRL-Kursor nach rechts (naechstes Wort)
117	CTRL-END
118	CTRL-PG DN
119	CTRL-HOME
120-131	ALT-1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, =
132	CTRL-PG UP