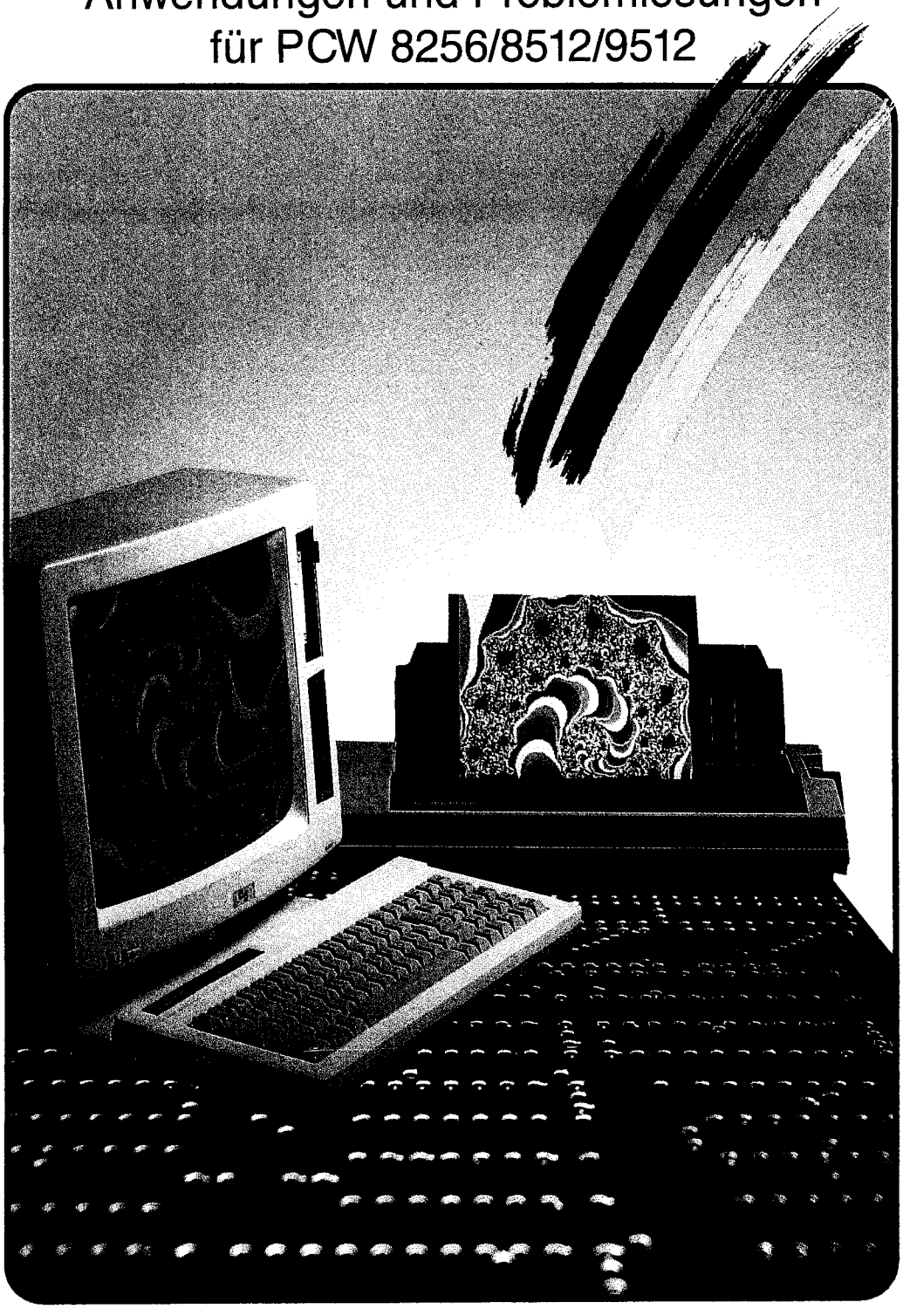
JOYCE

JOYCE - mehr als ein Textsystem

Anwendungen und Problemlösungen
für PCW 8256/8512/9512



DMV
Daten- und
Medienverlag

Mit großem Hardwareteil

Hinweis des Verlages:

Die in diesem Buch veröffentlichten Programme, Verfahren, Schaltungen etc. werden ohne Rücksicht auf eventuell bestehende Patente abgedruckt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt. Eine gewerbliche Nutzung ist nicht gestattet.
Alle Angaben und Programme dieses Buches werden vom Autor und vom Verlag sorgfältig überprüft. Dennoch lassen sich Fehler nie ganz ausschließen. Der Verlag kann daher weder die Gewähr für Fehlerlosigkeit noch die juristische Verantwortung oder irgendeine Haftung für eventuell auftretende Folgen übernehmen.

DMV-Verlag
Fuldaer Straße 6 · 3440 Eschwege · Tel. (0 56 51) 80 09-0
ISBN 3-926177-02-0 **Printed in Germany**

Im Jahre 1985 von der Firma Schneider auf den Markt gebracht, später von der deutschen Tochter des englischen Herstellers Amstrad weiter vertrieben, nahmen die PCWs - im Sprachgebrauch stets 'JOYCE' genannt - von Anfang an eine Sonderstellung unter den Computern ein. Mangels Farb- und Musikausstattung zum Spielen fast untauglich, durch die mitgelieferte Software und das geschlossene Hardwarekonzept schon von Seiten der Hersteller zur Textverarbeitung verdammt, ließen die PCWs ihre Besitzer über die mannigfaltigen Anwendungsmöglichkeiten stets im unklaren. Leser der in diesem Verlag herausgegebenen Monatszeitschrift 'PC International' konnten mitverfolgen, wie findige PCW-Besitzer die Geheimnisse ihres Computers nach und nach aufdeckten - mit der Zeit ergab sich eine stattliche Sammlung von Unterlagen zu den von Vertreiber, Hersteller und Händlern gern totgeschwiegenen Fähigkeiten der PCWs, die, wie bekannt ist, über Textverarbeitung weit hinausgehen. Die Autoren des vorliegenden Buches - in der 'JOYCE-Szene' keine Unbekannten - haben die bislang verfügbaren Informationen in mühevoller Kleinarbeit gefiltert und um wesentliche Teile ergänzt. Wenngleich es nicht den Anspruch erhebt, der Weisheit letzter Schluß zu sein, steht Ihnen mit diesem Buch doch die meines Erachtens vollständigste Zusammenstellung an Informationen über die 'unbekannte Seite' der PCWs zur Verfügung.

Für den Herausgeber
Eschwege, im April 1989
Michael Ebbrecht

Inhaltsverzeichnis

Vorwort

Seit Erscheinen der PCW auf dem deutschen Markt gab es auf dem Computersektor enorme Weiterentwicklungen, die für Preisstürze der "veralteten" Modelle verantwortlich waren. Wenn dennoch, wie die Verkaufsinserate verraten, so viele der über 100000 "PCW'ler" allein in der Bundesrepublik ihrem Joyce treu blieben, so liegen dafür handfeste Gründe vor.

Zunächst besticht der PCW durch die Einheit von Drucker, Tastatur und Monitor, die optimal aufeinander abgestimmt sind und einfachste Handhabung des mitgelieferten LocoScript garantieren. Dieses Textverarbeitungssystem war es auch, daß die meisten Anwender zu ihrer Kaufentscheidung bewog. Benutzerfreundliche Menueführungen gestatteten es, sofort nach dem Einschalten loszuschreiben, 90 Zeichen und 32 Zeilen (Auflösung von 720x256 Pixel!) am Monitor dargestellt zu bekommen, was längst nicht Standard ist, und ohne umständliche Anpassungen des Druckers gleich auszudrucken. Der PCW wurde als Textsystem angeboten und als solches gekauft. Wieviel jedoch der Joyce tatsächlich darüber hinaus leisten kann und daß er ein leistungsfähiger Computer ist, dem eine riesige internationale Bibliothek von CP/M-Software zur Verfügung steht, blieb vielen Anwendern verborgen.
Der PCW-Anwender braucht nicht einmal allzu hohe Zugeständnisse an die Geschwindigkeit seines Computers zu machen, da neuere Rechner meist auf ältere Software angewiesen sind, die für sie angepaßt wurde, und die deshalb die erreichbare Verarbeitungsgeschwindigkeit des Computers nicht ausnutzt.
Eines jedoch gibt es an diesem System zu kritisieren. Die mitgelieferten Handbücher erklären zum einen die integrierte Software höchst ungenau und unvollständig, zum anderen finden die Hardware und ihre Ausbaumöglichkeiten gar keine Erwähnung. Diesem Manko soll das vorliegende Buch entgegentreten und mithelfen, den Joyce vor den Augen seiner Anwender zu dem zu machen, was er de facto ist:

ein vollwertiger Computer.

Zu diesem Buch

Es gab in jüngster Vergangenheit bereits einige Publikationen, die dem Joycebesitzer eine wertvolle Hilfe sein können. Hier sind:

- die Artikel zum Joyce der Monatsschrift Amstrad/PC-International,
- das Buch "Praktische Textverarbeitung mit Joyce" von Jürgen Siebert,
- das CP/M Plus Anwender-Handbuch CPC 6128/Joyce von Jürgen Hückstädt,
- und das große Logo-Buch zu CPC und Joyce von Gerhard Sauer

zu nennen.
Andere Schriften zum Joyce sind weit weniger wertvoll und geben dem Anwender kaum die Hilfe und die Tips, die er von ihnen erwarten könnte.
Da gerade zum Betriebssystem CP/M und zur Textverarbeitung derart gute Publikationen vorliegen, die man nur zur Anschaffung empfehlen kann, möchten wir diese Themen (mit Ausnahme von MAIL232) nicht noch einmal aufgreifen und weiter ausbauen. Außerdem erschienen für den Joyce erweiterte LocoScript-Versionen und ein vollkommen neues Textverarbeitungssystem von Arnor (Prowort), das durch Schnelligkeit, integrierte Korrekturfunktionen und vieles mehr derart überzeugt, daß man kaum noch guten Gewissens das ein oder andere System durch Erwähnung bevorzugen kann.
Mit dem Buch von Gerhard Sauer liegt zwar eine gute Dokumentation zu "Logo" vor, doch scheint sich diese Sprache bei den Joyceanwendern nicht in dem Maße durchgesetzt zu haben, daß sich für sie die Anschaffung des Buches zu lohnen schien. Einsatz könnte diese Sprache zum Beispiel bei der Erstellung von Briefköpfen, kleineren Diagrammen oder schnellen Kalkulationen zwischendurch finden. Derartigen Anwendungen angemessen wurde ein entsprechend kurzes Kapitel dieses Buches auch der leider verkannten Sprache Logo gewidmet.
In erster Linie werden wir uns also um die Vertiefung und

Verfeinerung der Kenntnisse der Sprache BASIC und ihrer Jetsamverwaltung bemühen, wobei auch dem fortgeschrittenen Anwender wertvolle Tips an die Hand gegeben werden können.
Ebenfalls wichtig erschien uns, über Hardwareerweiterungen wie Scanner oder Akustikkoppler für den Joyce zu berichten. Nützliche Tips und Tricks für Basteleien an vorhandener Hardware dürfen in diesem Zusammenhang natürlich nicht fehlen.
Alles in allem hoffen wir mit vorliegendem Buch, dem Anwender ein erweitertes Einsatzspektrum seines Joyce zu eröffnen und ein effektiveres Arbeiten und Experimentieren mit diesem Computer zu ermöglichen.
Begleitend zu diesem Buch ist über den DMV-Verlag eine Diskette erhältlich, die sämtliche Listings (auch viele der kleinen Programme zum BASIC-Teil) "ready to run" beinhaltet. Darüber hinaus befinden sich auf ihr auch Programme, deren Listings im Buch zu viel Platz beansprucht hätten und für Erklärungen über Programmierung nicht mehr benötigt wurden. Dennoch handelt es sich um wertvolle Programme, die dem Anwender zusätzliche Informationen liefern können und sogar - wie das Programm "Generjet" - bei einer Programmierung an sich behilflich sein können. (Ein sogenanntes "tool").

Die im Buch veröffentlichten Platinenlayouts brauchen nicht selbst geätzt zu werden. Sie können über die Fa.:

Joyce-Platinenservice, Roesoll 36, 2305 Heikendorf

bezogen werden. Den Platinen wurde eine ausführliche Anleitung mit auf den Weg gegeben, so daß ein einigermaßen geschickter Bastler sie selbst bestücken kann. Dadurch, daß wir die Layouts selbst entwickelten und der Anwender sie selbst bestückt, konnten die Platinen recht preiswert auf den Markt gebracht werden (DM 49,-). Neben dem Sprachsynthesizer oder der computergesteuerten Lichterkette, die beide ein Novum für den Joyce darstellen, bietet sich mit der Schnittstelle eine echte Alternative zur üblichen Hardwareerweiterung.

Die Sprachen des Joyce

In den folgenden Kapiteln wird in erster Linie auf die Sprachen Logo und BASIC eingegangen. Dies liegt nahe, da sie jedem Anwender auf den Systemdisketten zur Verfügung stehen. Wer in diesen Sprachen zu programmieren weiß, Kreativität und bestimmte geistige Leistungsfähigkeit an ihnen erprobt hat, dem fällt es auch leicht, auf andere Sprachen umzulernen.
Daß die Sprache Logo z.B. nicht als "Exotensprache" im luftleeren Raum steht, beweist die Tatsache, daß es als Abkömmling von LISP zu sehen ist. LISP entstand in den frühen sechziger Jahren am Massachusets Institut of Technology speziell um die Möglichkeiten künstlicher Intelligenz zu studieren. Seymore Papert (ein Schüler Jean Piagets), der an diesem Institut arbeitete, entwickelte dann in langjähriger Arbeit die Sprache Logo, die man als ausgefeilten Dialekt von LISP sehen kann.
Dennoch sollen zu Ende des Kapitels noch einige Bemerkungen zu anderen Sprachen fallen. Eine beachtliche Zahl von Joyce-Programmierern arbeitet mit Turbo-Pascal. Anwender, die sich über diese Sprache noch nicht informieren konnten, werden hier ein kleines Review vorfinden, das kurz Vor- und Nachteile dieser Sprache aufzeigt.
Daß die Sprache C, die auf vielen Großrechenanlagen und Rechnern neueren Typs in zunehmendem Maße eingesetzt wird, auch für den Joyce erhältlich ist, war vielen Joyce-Usern unbekannt. Welche immensen Vorteile hinter dem Systemangebot C-Compiler - zudem noch aus deutschen Landen - stecken, wird zu Ende des Kapitels Erwähnung finden.
Forth, als Public Domain Software für den Joyce erhältlich, scheint aus dem Rennen der Gunst der Programmierer geschlagen zu sein. Vor dem Einstieg in die Arbeit mit Logo steht an dieser Stelle also nur der Hinweis, daß der Joycer selbst darauf nicht zu verzichten braucht.
Zur Schreibweise: Sind Tasten der Konsole gemeint, so werden sie in eckige Klammern und in große Lettern gesetzt. Variablen und Listingauszüge finden sich in Fett- bzw. Kleindruck.

Logo

Allgemein:

In der Zeit, in der der Joyce auf dem Markt ist, hat sich herausgestellt, daß sich die meisten seiner Anwender auf die Benutzung des mitgelieferten LocoScript beschränken, oder auf zugekaufte Programmsysteme zurückgreifen. Weitaus seltener versuchen sie sich darin, selbst zu programmieren. Dabei liegt gerade mit Logo eine Sprache vor, die für Einsteiger besonders geeignet scheint, und zum Experimentieren einlädt.
Diese Eigenschaft der Sprache hat ihr zu Unrecht ihren schlechten Ruf als "Kindersprache" eingebracht. Dabei sind Sprachen wie BASIC oder Pascal nicht so ohne weiteres - zumindest nicht ohne elementare Kenntnisse über die GSX-Schnittstelle - dazu zu bewegen, Grafik auf den Schirm zu bringen. Für den Logo-Anwender ist dies allerdings kaum ein Problem, wurde Logo doch gerade durch seine Turtle-Grafik bekannt.
Wer also in erster Linie seine Sekretärin Joyce zum Schreiben nutzt, sollte doch ruhig einmal probieren, wie man sie auch zum Zeichnen kleinerer Grafiken oder zum Errechnen und Aufzeichnen von Diagrammen bewegen kann. Bestimmt wird dies auch der erste Schritt zu komplexen Programmstrukturen werden, die von dreidimensionaler Grafik bis hin zur Listenverarbeitungen reichen können.

Die Initialisierung:

Die Sprache Logo hat bei den nicht professionellen Joyce-Anwendern auch immer ein wenig im Schatten gestanden, weil die Dokumentation in Form des Handbuches kaum ihren Namen verdient. Viele Befehle, die Logo kennt und ausführen kann, werden überhaupt nicht erklärt, und Befehle wie **po**, **ss** oder **setsplit** werden gleich mehrfach erörtert.
Die meisten Handbücher sagen nicht einmal, wie Logo denn zu

starten sei, was an dieser Stelle geschehen soll.
Für den Anwender, der mal schnell zwischendurch in Logo zeichnen oder rechnen möchte, empfiehlt es sich, eine Startdiskette anzufertigen, die nur ins Laufwerk geschoben werden muß und Logo automatisch startet. Am zweckmäßigsten kann dies mit Hilfe von LocoScript bewerkstelligt werden. Zunächst werden die Files:

J14GCPM.EMS
LOGO.COM
KEYS.DRL
SUBMIT.COM
LANGUAGE.COM
SETKEYS.COM

von den Seiten 2 und 4 der Systemdisketten in eine Gruppe von Laufwerk M: in LocoScript geladen. Danach wird mit "E" unter LocoScript eine Datei erstellt, die nichts weiter enthält, als den genau so geschriebenen Text:

```
LANGUAGE 0
SETKEYS KEYS.DRL
LOGO.COM
```

Diese Datei muß jetzt als ASCII-Datei (eigener Menuepunkt unter LocoScript) gespeichert werden. Dabei fragt Joyce nach dem Namen, den diese Datei erhalten soll. Er wird mit "PROFILE.SUB" angegeben und ebenso wie die vorher in M: gespeicherten Dateien in die Gruppe 0 des Laufwerks A:, welches mit einer neu formatierten Diskette bestückt wurde, abgespeichert.
So befinden sich die Dateien nun auf einer neuen Startdiskette in Laufwerk A:. Nach dem gleichzeitigen Drücken von [SHIFT]-[EXTRA]-[EXIT] müßte der Monitor dunkel werden, das Laufwerk gutwillig brummend seine Arbeit aufnehmen und das Betriebssystem booten. Das Betriebssystem arbeitet dann die PROFILE.SUB-Datei der Reihe nach ab, schaltet auf den amerikanischen Zeichensatz um, damit z.B. die Ä's als eckige

Klammern auf dem Monitor erscheinen, ändert die Tastaturbelegung, damit z.B. Drücken von [F1] pausing unter Logo verursacht und ruft dann Logo selbst auf. Am Ende des Prozesses steht uns Logo mit seinem ?-Prompt zur Verfügung.
Wer wirklich professionell mit Logo arbeiten möchte, sollte auch die Standardeinstellung des Druckers ändern, damit in den Listings ebenfalls eckige Klammern und dergleichen erscheinen. Diese Änderung geschieht am zweckmäßigsten mit dem Programm SETLST.COM von Seite 2 der Systemdisketten. SETLST.COM wird ebenfalls auf die Startdiskette kopiert. Unter LocoScript wird wieder ein ASCII-File mit dem Namen "DRUCKER.DRL" erstellt, der nur die Zeile:

```
↑'ESC'R↑'0'
```

enthält. Auch diese Datei wird auf der Startdiskette abgespeichert. In die POFILE.SUB-Datei braucht nur noch die Zeile:

```
SETLST DRUCKER.DRL
```

vor der Zeile LOGO.COM eingefügt werden, so daß bei einem Neustart der Drucker automatisch richtig eingestellt wird.
Auf der Startdiskette selbst ist jetzt noch genügend Platz, um etliche Prozeduren, die den Wortschatz der Sprache erweitern sollen, zu speichern. Einer effektiven Arbeit mit Logo steht jetzt nichts mehr im Wege.

Logo als Rechner

Für viele Aufgabenbereiche kann es durchaus sinnvoll erscheinen, Logo auf dem Joyce zu booten und schnell einige Rechnungen durchzuführen. Mal abgesehen von der Leistungsfähigkeit des Systems, an die nur Taschenrechner der gehobenen Klasse heranreichen, kann der Drucker sämtliche Aktionen des Rechners protokollieren. Dazu wird lediglich hinter das ?-Prompt von Logo

```
copyon
```

eingegeben, und nachdem [RETURN] gedrückt wurde, werden sowohl Aufgaben als auch Lösungen oder Ergebnisse zu Papier gebracht. Fast könnte man meinen, Joyce kopiere eine Registrierkasse.
Wird ein Protokoll nicht mehr gewünscht, reicht die Eingabe von

```
copyoff
```

um den Drucker seine Arbeit einstellen zu lassen. Ein weiterer Vorteil des Systems besteht darin, daß man häufig gebrauchte Konstanten wie Kreiszahl pi, den Mehrwertsteuersatz oder den Skontorabatt einfach eingeben und abrufen kann und daß häufig durchzuführende Rechnungen in Prozeduren festgelegt werden können, die dann schnell zur Hand sind. Eine solche Festlegung kann in Logo so aussehen:

```
make "Mwst 0.14
```

Bei diesem Befehl gilt es zu beachten, daß die Leerstellen und die Anführungszeichen gesetzt werden, um keine Fehlermeldung folgen zu lassen. Außerdem sollte man sich merken, ob Mwst oder MWST oder mWst definiert wurden. Das sind für Logo drei verschiedene Vokabeln! Nachdem nun die Mwst für Logo festgelegt wurde, kann man auch damit rechnen. Ein simples Beispiel würde so aussehen:

```
50 * :Mwst
```


Hier wird Mwst jetzt mit einem vorangestellten Doppelpunkt gekennzeichnet. Die unterschiedlichen Notationen in Listings bedeuten:
-Das einleitende Anführungszeichen zeigt Logo, daß es sich bei dem Folgenden um einen Platzhalter oder ein freies Fach für etwas handelt.
-Der einleitende Doppelpunkt zeigt Logo, daß hier der Wert, für den der Platzhalter steht, eingesetzt werden muß.
-Keine Bezeichnung läßt Logo vermuten, daß eine Prozedur oder Kommando gemeint ist. Logoeigene Befehle werden immer klein geschrieben.
Hier nun die möglichen Rechenoperationen:

Form	Ausgabe	Erklärung
2.1 + 3.4 oder:		Sollen + oder - als Vorzeichen dienen, so
+ 2.1 3.4	5.5	müssen sie ohne Leerstelle vor die be-
(†Dezimalpunkte !)		treffende Zahl gesetzt werden. (Subtraktion entsprechend)
3 * 2 oder: * 2 3	6	Multiplikation
8 / 2 oder: / 8 2	4	Division
Komplex:		
(1.4 + 6.6) / (4 - 1 * 2)	4	Punktrechnung vor Strichrechnung !
(+ 4 2 3 9 2)	20	Mehr als 2 Elemente müssen in runde Klammern gesetzt werden.
sin 30	0.5	Sinuswert für Winkel 30 Grad (Angaben nur in Gradmaß)
cos 30	0.866025388240814	Cosinuswert für Winkel 30 Grad. (In allen Logorechnungen sind nur die ersten 7 Stellen hinter dem Punkt verläßlich !!!)
arctan 0.5	26.565051177178	Arctanwert von 0.5 (Gradmaß)
round 2.2	2	Rundet auf ganze Zahl auf oder ab.
int 2.3	2	Gibt den ganzzahligen Wert einer Zahl an
quotient 10 4	2	Ganzzahliger Wert des Bruchs 10/4
remainder 10 4	2	Der Rest bei der Ganzzahldivision 10/4
1 + random 49	z.B. 16	Ermittelt eine Zufallszahl, die wegen 1+ größer als 0 und kleiner als 50 ist.Die Zufallsreihen sind nach jedem Neustart gleich. Sie können mit "rerandom" wieder zurückgesetzt werden.

Basierend auf diesen Grundfähigkeiten von Logo lassen sich alle anfallenden Probleme lösen. Was Logo nicht kann, läßt sich als Prozedur definieren und in seinen Wortschatz einbauen. Dies läßt sich mit dem Editor von Logo einfach bewerkstelligen. Nach Eingabe von **ed** geht Logo in einen

anderen Modus über, und man kann das Geschriebene wie in einem Textsystem löschen, mit [DEL] korrigieren, oder mit den Cursortasten darin herumfahren.
Ein einfaches Programm müßte jetzt so eingetippt werden:

```
to kreisumfang :r
make "pi 3.1415926
op 2 * :pi * :r
end
```

Die erste Zeile gibt Logo an, wie diese Prozedur in Zukunft zu heißen hat. Wenn man später nicht so viel tippen will, könnte sie auch kr heißen. Mit dem **:r** signalisiert man Logo, daß in Zusammenhang mit der Prozedur "Kreisumfang" eine Eingabe - in diesem Fall der Radius - zu machen sein wird, die Logo dann auch unbedingt beim Aufruf erwartet.
Die zweite Zeile müßte hier nicht stehen. Hätte man Logo außerhalb dieser Prozedur den Wert für **pi** gegeben, wie vorhin beim Beispiel **Mwst**, könnte er ihn sich auch für viele andere Aufgaben aus seinem Arbeitsspeicher holen.
Die dritte Zeile beinhaltet die eigentliche Rechnung, deren Ergebnis als **op** (Output) an den Monitor gegeben wird. Die vierte Zeile signalisiert Logo, daß die Prozedur hier beendet ist. Sollte man sich bei Eingabe dieser Prozedur sehr vertippt haben und man möchte lieber noch mal von Vorne anfangen, so genügt ein Druck auf die [STOP]-Taste. Ist alles richtig eingetippt, wird die [EXIT]-Taste gedrückt, Logo verläßt den Editier-Modus und meldet: kreisumfang defined. (Der Umgang mit [STOP]- und [EXIT]-taste wird in älteren Handbüchern verwechselt!)
Nach der Eingabe von:

```
kreisumfang 5
```

wird Logo den Umfang eines Kreises mit dem Radius 5 berechnen und das Ergebnis auf den Schirm bringen. Hier wird eigentlich noch nicht ersichtlich, inwieweit eine Arbeitserleichterung gegeben ist, es sei denn, jemand müßte am laufenden Band Kreisumfänge errechnen. Ziel vorliegenden Buches ist es nun nicht, einen Kurs über Programmieren in Logo zu liefern. Es sollen nur die Möglichkeiten kurz

aufgezeigt werden. Ein schönes Beispiel für eine komplexere Struktur liefert die Errechnung der Quadratwurzel. Aus unverständlichen Gründen hat Logo sie nicht in seinem Grundwortschatz, und so muß sie als Prozedur extra definiert werden. Nach dem Aufruf von **ed** gibt man ein:

```
to sqrt :x
op wurz (:x + 1) * 0.5
end
to wurz :a
local "b make "b ((:x /:a + :a) * 0.5)
if :a - :b < 1.e-03 [op :b]
op wurz :b
end
```

Nach dem Drücken von [EXIT] erscheint: Sqrt defined wurz defined. Hier wurden zwei Prozeduren definiert, die aber beim Abspeichern auf Diskette mit:

```
save "sqrt
```

beide gesichert würden, und beim Laden von Diskette in den Arbeitsspeicher mit:

```
load "sqrt
```

beide im Wortschatz zur Verfügung stünden. Logo lädt und sichert zusammengehörige Prozeduren gemeinsam. Die erste Prozedur nun liefert die Hälfte des um eins vermehrten Radikanden durch **op** (Output) an die zweite Prozedur, wobei dieser Rest den Wert **:a** erhält. In der zweiten Prozedur wird jetzt der Radikant durch **:a** geteilt, um **:a** vermehrt und um die Hälfte verkleinert. Was dabei herauskommt muß Logo jetzt als Wert **:b** sehen. Als nächstes schaut Logo in der if-Zeile, ob die Differenz zwischen **:a** und **:b** schon kleiner als 1.e-03 ist. Trifft dies zu, kann Logo die erste in eckigen Klammern stehende Befehlsliste ausführen, also den Wert von **:b** als Ergebnis ausgeben. Trifft es nicht zu, geht alles noch mal bei **wurz** von vorne los, wobei **:b** der neue Anfangswert von **wurz** ist.
Local "b in der zweiten Prozedur zeigt Logo, daß der Wert, der im Folgenden ausgerechnet und durch **make** der Variablen **"b** zugeordnet wird, nur für diesen einen Durchlauf der

Prozedur Gültigkeit hat und bei jedem weiteren Durchlauf neu zugemessen werden kann.
Bei seiner Rechnerei läßt sich Logo auch in die Karten schauen. Tippt man hinter das ?-Prompt

```
trace
```

ein, so zeigt Logo die Rechenergebnisse aller Zwischenschritte. Noch genauer kann man die Sache verfolgen, wenn

```
watch
```

eingegeben wird. Danach läßt sich Logo mit einer neuen Wertzuweisung solange Zeit, bis [RETURN] gedrückt wurde. Außerdem wird der Rechenschritt, bei dem er sich gerade befindet, genau aufgezeichnet. Durch

```
notrace und nowatch
```

hinter dem Prompt, wird dieser Zustand wieder abgeschaltet.
Eigentlich müßte man für den Fall, daß jemand eine Null oder negative Zahl radizieren will, eine Sicherung in die Wurzelfunktion einbauen. Dies kann auch wieder durch eine **if**-Zeile geschehen. In der zweiten Zeile der **sqrt**-Prozedur könnte dann stehen:

```
if :x = 0 [op 0] if :x < 0 [op "neg.]
```

Es lohnt sich schon tatsächlich, eine Wurzelfunktion im Arbeitsspeicher Logo's zu haben, zumal sie auch für den eigentlichen Arbeitseinsatz von Logo, der Grafikprogrammierung, gut zu gebrauchen ist. An späterer Stelle wird sie noch einmal auftauchen, wenn demonstriert werden soll, wie Logo nach dem Satz des Pythagoras Dreiecke berechnet und konstruiert.
Sollte man sich bei der Eingabe der Wurzelfunktion ohne es zu merken vertippt haben, so wird Logo nach dem Aufruf der Prozedur eine Fehlermeldung auf den Monitor bringen. Tippt man nun gleich den Befehl **ed** ohne weitere Spezifikation hinter das Prompt, so geht Logo mit der fehlerhaften Prozedur in den Editiermodus über, und läßt den Cursor an

der fehlerhaften Stelle blinken.
Dies mag als Einblick in die Rechenfähigkeiten Logo's genügen und als Überleitung zur Beschäftigung mit der Grafik dienen.

L o g o a l s G r a f i k e r

Vorweg:

Anwender, die nur ein wenig mit Logo herumspielten und einige kleine Grafiken auf den Schirm zaubern konnten, hatten, wenn sie nicht tiefer in die Materie eingestiegen waren, zwei Probleme:
Zum einen schweigt sich das Handbuch darüber aus, wie man eine Hardcopy des Bildschirms zu Papier bringen kann, - nämlich mit gleichzeitigem Drücken von:

[EXTRA]/[PTR]

und wenn sich das durch Mund- zu Mundpropaganda herumgesprochen hatte, so bekam man - wenn es nicht gelang, durch Druck auf [PTR] rechtzeitig den Vorgang abzubrechen - zusätzlich zu seiner Grafik noch die Meldung aufs Papier gedruckt, daß A:das aktuelle Laufwerk sei. Dieser Mißstand läßt sich dadurch beheben, daß hinter das Logo-Prompt eingetippt wird:

```
type word char 27 "0
```

Nach Druck auf [RETURN] verschwindet diese lästige Anzeige. Wiederherstellen läßt sie sich durch Eintippen der 1 anstelle der 0. Mit dem Befehl **type word char 27 "**(+Ziffer oder Zahl) läßt sich noch mehr anstellen, wie im Abschnitt über Beschriftung von Grafik ersichtlich wird. Neugierigen sei an dieser Stelle zunächst empfohlen, sich die Escape-Folgen in diesem Buch anzusehen. Unter Logo wird nur für ESC

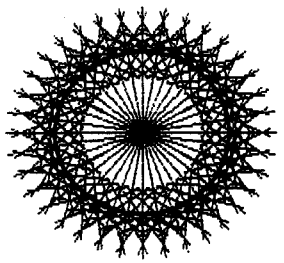
das **type word char 27 "** eingesetzt. Nach diesen einleitenden Bemerkungen aber nun zur eigentlichen Grafikprogrammierung.

Die freie Turtlebewegung

Die Überschrift macht hier schon kenntlich, daß der Zeichencursor sich auch unter anderen Grundbedingungen als frei auf der Fläche des Monitors bewegen kann. Gemeint ist hier ein Koordinatensystem, das Vorgaben für die Turtlebewegung geben kann. Dazu aber später mehr. Hier interressieren erst einmal die Befehle, die dem Grafikcursor an seiner Position relativ zu seiner Stellung Anweisungen geben.
In der kurzen erklärenden Einleitung des Handbuches werden die grundlegenden Formen von Eingaben z.B. mit **repeat** gezeigt. Prinzipielle Formen der Eingabe brauchen deshalb nicht mehr erklärt werden. Interessant wäre es aber einmal, eine Verschachtelung von Repeat-Anweisungen zu zeigen. Dies könnte etwa so aussehen:
(Die Eingabe erfolgt in einer Zeile, wird die Zeile für die Monitordarstellung zu lang, macht Logo automatisch einen Zeilenumbruch, der durch ein Ausrufungszeichen zu Ende der Zeile kenntlich gemacht wird, darum braucht man sich nicht zu kümmern)

```
cs ct fs ht repeat 36 [fd 150 repeat 10 [fd 100 bk 100 rt 36] bk 150 rt 10]
```

Nach Betätigen von [RETURN] entsteht folgendes Gebilde:



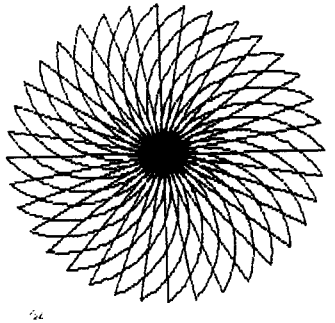
Nachdem der Schirm mit **cs** (clear screen) und **ct** (clear text) gesäubert wurde, wurde dem Grafikcursor mit **fs** (full screen) der vollständige Monitor zum Malen zur Verfügung gestellt. Mit **ht** (hide turtle) wurde der Cursor unsichtbar gemacht, was ein schnelleres Zeichnen zur Folge hat. Danach muß der Cursor die zwei Wiederholungsanweisungen durcharbeiten. In der ersten werden Strahlen der Länge 150 in 36 Schritten zu 10 Grad gelegt, was eine volle 360 Grad Drehung ausmacht, in der zweiten Prozedur werden um die Spitzen der Strahlen noch einmal Strahlen der Länge 100 gelegt. Hier ist die Anzahl allerdings geringer, es sind nur 10, die jeweils 36 Grad auseinander liegen.
Durch Variation der Längen und Gradeinteilungen lassen sich so interessante geometrische Gebilde erzeugen. Die bisherigen Anweisungen reichen auch aus, um die Turtle auf Kurvenbahnen zu schicken. So würde die Eingabe von:

```
repeat 50 [fd 4 rt 2]
```

eine Linie erzeugen, die 50 Mal nach jeweils 4 Schritten eine Rechtsdrehung vornimmt. Abstände und Winkel sind dabei so klein, daß der Eindruck einer Kurve entsteht. Auch diese Anweisung läßt sich in komplexere Prozeduren einbauen. So würde:

```
to t :a
repeat 50 [ fd 4 rt 2 ] home rt :a
local "b make "b (:a + 10) if :b > 360 [stop]
t :b
end
```

obige Prozedur - aufgerufen mit: **cs ct fs t 10** - die Rosette auf der folgenden Seite aus Kreissegmenten produzieren. Der Befehl **home** läßt die Turtle wieder in ihre Ausgangsposition zurückkehren, wobei sie eine Linie hinter sich herzieht und ihre Spitze wieder nach oben zeigt. Damit das nächste Rosettensegment nun um 10 Grad gegenüber dem vorherigen versetzt ist, muß sie nach dem home-Befehl eine Drehung nach rechts machen, die nach jedem Segment um 10 Grad größer ist als die vorherige. Deshalb wird **b** nach jedem Durchgang um 10 vergrößert und **t** wieder mit dem neuen **b**-Wert

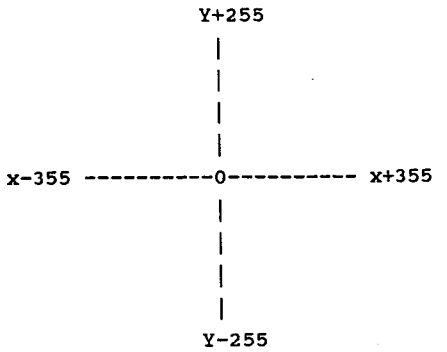


aufgerufen. Logo setzt den errechneten Wert von **:b** automatisch im folgenden Durchgang als **:a** ein. Damit dieser Prozeß nicht ad infinitum läuft, wurde mit **if** wieder ein Abbruch definiert. Sobald bei 360 Grad der Kreis geschlossen ist, stoppt Logo die gesamte Prozedur.
Mit Verknüpfung von Vorwärtsschritten und Rechtsdrehungen läßt sich nun eine Prozedur definieren, die nach Eingabe des Radius einen Kreis zeichnet. Um diese Prozedur anwenderfreundlicher zu gestalten, wäre es von Vorteil, den Cursor nach dem Zeichnen eines Kreises wieder auf seine Ausgangsposition zu dirigieren. An diesem Teil der Prozedur kann man sehr schön die Arbeit mit dem Koordinatensystem erklären, womit auch gleich die Überleitung zum nächsten Kapitel gegeben wäre.

Die Turtlebewegung im Koordinatengitter

Für die Positionierung des Grafikcursors mit Hilfe von Koordinatenpunkten stehen eine x-Achse mit 710 und eine y-Achse mit 510 Skalenpunkten zur Verfügung. Diese Werte beziehen sich auf den mit **fs** eingerichteten vollen Grafikschirm, wobei tatsächlich jeweils einige Grafikpunkte pro Koordinate mehr zur Verfügung stehen, aber bei der Arbeit mit Extremwerten bestünde Gefahr, den Cursor aus dem sichtbaren Bereich zu steuern. Der Nullpunkt der Skalen

liegt in der Bildschirmmitte, wie die erläuternde Grafik zeigt.
Logo kennt nun einige Befehle, die mit diesen Koordinaten arbeiten, wobei der zuerst genannte Wert sich immer auf die x-Achse bezieht.



setpos [-355 255]
würde den Grafikcursor in die linke obere Ecke steuern,
dot [0 0]
in die Mitte einen Punkt malen,
setx 10
würde die Turtle auf der x-Achse auf den Punkt 10 setzen,
sety 10
diese entsprechend auf die y-Achse.
seth towards [100 100]
würde die Spitze des Grafikcursors auf diesen Punkt ausrichten. (seth 90 bewegt die Turtle nicht im Koordinatensystem, sondern bewegt die Turtle wie unter dem Befehl rt, nur daß die Gradzahlen hier wie Himmelsrichtungen zu sehen sind: 0 = Nord, 90 = Ost, 180 = Süd, 270 = West etc.

Man kann allerdings auch Fragen stellen, die mit oder über Koordinaten beantwortet werden. So antwortet Logo auf **tf**

(turtle facts) mit einer Liste, von der die beiden ersten Zahlen die Koordinatenpunkte wiedergeben, auf denen sich der Cursor befindet. (Der Rest bezeichnet der Reihe nach: Richtung, in die der Zeichenstift zeigt (s. **seth**), Zustand des Zeichenstifts **pu** = abgehoben, **pd** = Zeichenstift unten, **pe** = Zeichenstift radiert, **px** = radiert oder zeichnet im Gegensatz zum Untergrund. Die vorletzte Zahl gibt die Farbe des Zeichenstifts an (0 oder 1) und die letzte, ob die Turtle sichtbar ist = true oder nicht = false).
Befände sich auf den Koordinaten 100 100 ein gesetzter Punkt, so würde

```
dotc [100 100]
```

mit der Ausgabe 1 antworten; wäre diese Stelle nicht belegt, so würde 0 ausgegeben. Dotc fragt also nach, ob ein Punkt gesetzt ist, oder nicht.
Um für die Kreisprozedur den Ausgangspunkt festzulegen, zu dem der Grafikcursor nach verrichteter Arbeit wieder zurückkehren soll, fehlt noch eine Vokabel. Wie besprochen liefert die Frage nach den Turtle-facts mit **tf** eine Liste, von der aber für **setpos** oder **setx/sety** nur die ersten zwei Werte benötigt werden. Um diese beiden Werte aus der Liste "herauszufiltern", können **piece** (wird z.B. im Handbuch nicht erklärt) und **item** eingesetzt werden.

```
piece 2 4 tf
```

würde aus der Liste der Turtle-facts das zweite bis vierte Element ausgeben. **Piece** ist also in der Lage, mehrere nebeneinanderliegende Elemente einer Liste auszuwerfen. **Item** ist nur in der Lage, ein einzelnes Element auszugeben. So würde

```
item 2 tf
```

genau das zweite Element dieser Liste zur Ausgabe haben. Die Belegung einer Variablen mit der derzeitigen Cursorposition könnte also so aussehen:

```
local "a make "a (piece 1 2 tf)
```


Will man später den Cursor auf diese Position setzen, so reicht:

```
setpos :a
```

um ihn dorthin zu dirigieren.
Arbeitet man mit Variablen für die Koordinatenpunkte, so muß aus ihnen vor der Verwendung in setpos eine Liste erstellt werden. Dies kann mit **se** bewerkstelligt werden. Als Befehl sähe das so aus :

```
setpos se :x :y
```

Zurück jedoch zur Kreisprozedur. Als einzige Eingabe soll die Prozedur "Kreis die Eingabe **:r**, den Radius, benötigen. Nach Feststellung und Merken des Standortes durch **make** und **piece/tf** muß die Turtle den so festgelegten Kreismittelpunkt auf gerader Linie um den Radius **:r** verlassen, eine Rechtsdrehung machen und dann eine Strecke wandern, die sich aus dem Umfang - also (2*pi*r) errechnet. Diese Strecke wird z.B. 100 mal durch eine Rechtsdrehung unterbrochen. Da bei den Drehungen insgesamt 360 Grad herauskommen müssen, muß jede einzelne der 100 Drehungen 3.6 betragen. Die Turtle hat folglich 100 mal den hundertsten Teil des Umfangs zu laufen und sich zwischendurch immer um 3.6 nach rechts zu drehen. Wenn sie fertig ist, kann sie sich, durch **setpos** dorthin gelenkt, wieder auf dem Kreismittelpunkt ausruhen. Als Prozedur sieht das folgendermaßen aus:

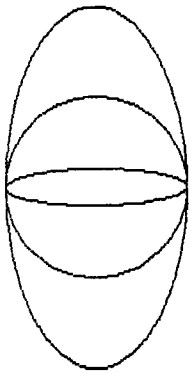
```
to Kreis :r
(local "a "b)
make "pi 3.14159265
make "a (piece 1 2 tf)
pu fd :r rt 92 pd
make "b 2 * :pi * :r / 100
repeat 100 [fd :b rt 3.6]
pu setpos :a lt 92 pd
end
```

Über die Kreisprozedur lassen sich auch Ellipsen konstruieren. Gibt man Logo mit **sf** die Frage nach den screen-facts, den Bildschirmzuständen, ein, so erhält man eine Liste mit 5 Elementen. Daraus kann man der Reihe nach

die Hintergrundfarbe des Grafikfensters, ob **ts**, **ss** oder **fs** aktiviert sind, die Fensteraufteilung, seine Aufteilung durch **fence**, **window** oder **wrap** und als letztes das Achsenverhältnis erfahren. Standartmäßig ist dieses Verhältnis mit 0.46875 ausgelegt. Es kann aber mit **setscrunch** geändert werden. Wie **setscrunch** genau arbeitet, läßt sich am besten feststellen, wenn folgender Befehl eingegeben wird:

```
cs ct fs ht setscrunch 1 Kreis 100 setscrunch 0.5 Kreis 100 setscrunch 0.1 Kreis 100
```

Damit werden der Reihe nachverschiedene Ellipsen erzeugt. Wird **setscrunch** mit 0.49625 eingestellt (Verhältnis 1 zu 1), entsprechen bei einer Hardcopy sowohl auf der X- wie auf der Y-Achse 5.7 Logoeinheiten genau einem Millimeter auf dem Papier!
Im Kapitel über das Rechnen mit Logo wurde eine Dreiecksprozedur angekündigt, die mit Hilfe der Wurzelfunktion im rechtwinkligen Dreieck nach dem Satz des Pythagoras die fehlende Seite ausrechnet und zeichnet. Es wäre nützlich, wenn Logo das Ergebnis seiner Rechnung unter die Zeichnung stellen würde. Die Realisierung dieses Problems führt uns in das nächste Kapitel, wo erläutert wird, wie Schrift in Grafiken gesetzt wird.



Die "setscrunch-Ellipsen"

Beschriftung von Grafik

In diesem Kapitel sollen sämtliche Fähigkeiten des Systems zum Tragen kommen. Es soll also ein Programm entwickelt werden, daß nach gewissen Vorgaben etwas ausrechnet, eine kleine Grafik dazu auf den Schirm bringt und das Ergebnis unter diese Grafik stellt. Dafür bietet sich die Dreiecksberechnung an. Das Problem gestaltet sich nicht zu komplex und bietet genügend Raum für Erklärungen.
Sind zwei Seiten im rechtenwinkeligen Dreieck bekannt, so läßt sich nach a↑2+b↑2=c↑2 die fehlende Seite berechnen. Bekannt seien in unserem Fall die Seiten a und b. Es wird also wieder die bereits erstellte Wurzelfunktion benötigt, denn c ist ja dann:

```
make "c sqrt (:a * :a + :b * :b)
```

Das Prozedere soll nun sein, daß der Grafikcursor von einem Punkt, den er sich merkt, die Strecke **:a** läuft, eine Rechtsdrehung um 90 Grad vollzieht, die Strecke **:b** läuft und dann die Spitze seines Cursors mit **seth towards** auf den gemerkten Ausgangspunkt dreht. Er braucht dann nur noch mit der errechneten Strecke **:c** zu laufen, um das Dreieck zu vollenden. Als Prozedur sähe das dann so aus:

```
to Dreieckab :a :b
cs ct fs ht
(local "h "c)
make "h (piece 1 2 tf)
make "c sqrt ((:a *:a)+ (:b*:b))
rt 45 fd :a rt 90 fd :b seth towards :h fd :c
```

(Eine weitere elegante Lösung bei Dreieckskonstruktionen, auf die ich hier aber nicht weiter eingehen möchte, bietet sich darin, daß man Halbreise schlagen lassen und mit **dotc** und **if**-Bedingung Schnittpunkte abfragen kann, die dann als unbekannte Ecke des Dreiecks angesprochen werden können.)
Jetzt bleibt in unserem Beispiel oben noch das Problem, daß der Textcursor unter das gezeichnete Dreieck gesteuert werden muß, um die Länge von **:c** auszudrucken. Ebenso wie bei

setpos wird der Textcursor über Koordinatenpunkte gesteuert und als Eingabe zu dem Befehl:

```
setcursor [:x :y]
```

wird eine Liste erwartet. Hier jedoch bezeichnet der Wert von x eine Spalte (0-88) und y eine Zeile (0-29). **setcursor [0 0] type [hier]** würde in der linken oberen Ecke das Wort "hier" erscheinen lassen, **setcursor [88 29] type [hier]** würde das Wort in der rechten unteren Ecke des Monitors sichtbar machen. Um also den Textcursor nach den Turtlekoordinaten auszurichten, müßte man eine Umrechnungstabelle für die verschiedenen Koordinatensysteme anlegen. Da jedoch die Dreiecksprozedur mit **cs** und **ct** gestartet wird, landet der Grafikcursor wieder in der Bildschirmmitte. Würden wir also den Textcursor mit **setcursor [45 20]** dirigieren, so käme er auf jeden Fall unter dem gezeichneten Dreieck zu stehen. Hinter dem Befehl zur Plazierung des Cursor käme dann zu stehen, was er dort machen soll, nämlich **c=** und den Wert von **c** auf den Schirm zu drucken, also:

```
type [c =\ ] type :c
```

(Wegen \ wird der nachfolgende Zwischenraum gedruckt.Bei richtiger Tastaturbelegung liegt der Backlash auf Taste [F3])
Ein Problem bleibt noch offen. Die Dreiecksprozedur wurde mit **fs** gestartet, um auch größeren Dreiecken auf dem Grafikmonitor Platz zu bieten und um bei einer Hardcopy auf dem Drucker den Textcursor nicht zu sehen. Soll jetzt der Textcursor etwas auf den Schirm bringen, muß kurzzeitig mit **ts** (text screen) auf diesen Status umgeschaltet werden. Nachdem der Textcursor seine Arbeit verrichtet hat, kann wieder auf **fs** geschaltet werden. Die vollständige Zeile, die am Ende der Dreiecksprozedur eingesetzt werden muß, sieht also so aus:

```
ts setcursor [45 20] type [c=\ ] type :c fs
```


Zum Abschluß der kleinen Einführung möchte ich auf verschiedene Arten der Beschriftung von Grafiken, die schon nicht mehr so sehr nach mathematisch definierten Gebilden aussehen und in denen sogar Zufallszahlen eine Rolle spielen, eingehen.
Zunächst macht es immer einen guten Eindruck, wenn Grafiken umrandet sind. Eine Hardcopy sieht dann gleich viel gewinnender aus. Den Rahmen können wir mit **setpos** anfertigen. Zwei Vierecke werden ineinander gelegt und der Zwischenraum mit dem Befehl:

```
fill
```

aufgefüllt. Dieser Befehl ist mit Vorsicht zu genießen. Ist ein Zwischenraum nicht vollständig geschlossen, wird der gesamte Monitor ausgefüllt, und dieser Vorgang läßt sich nur durch brutales Ausschalten des Computers unterbrechen. Außerdem darf die Turtle nicht auf einem belegten Bildschirmpunkt stehen. Sie muß mit **pu** in die zu füllende Fläche gesetzt und danach mit **pd fill** in Aktion gesetzt werden. Nun zum Rahmen:

```
to Rahmen
pu setpos [-355 255] pd rt 90
repeat 2 [fd 710 rt 90 fd 510 rt 90]
pu setpos [-350 250] pd
repeat 2 [fd 700 rt 90 fd 500 rt 90]
pu setpos [-352 252] pd fill pu home pd
end
```

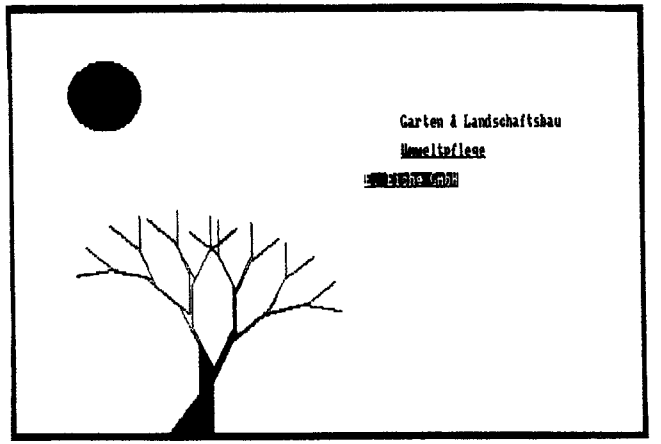
Nun zu dem, was in den Rahmen hinein soll,: ein Baum. Bei unserem Baum kann man in Grenzen mit Zufallszahlen arbeiten, wobei nach dem **fill**-Befehl des ganzen Bildes recht unterschiedliche Exemplare zustande kommen können. Für den Baum werden zwei Werte für Geraden **g** und Drehungen **d**, und ein Wert als Zähler für die Verschachtelungstiefe **v** gebraucht. Bei den Aufrufen der Baumprozedur werden die Strecken kontinuierlich gekürzt (Faktor 0.9), so daß der Eindruck von Ästen entsteht. Gelaufene Strecken werden genauso zurückgegangen, wobei durch die random-Zahl leichte Verschiebungen auftreten, die sich nach unten hin verbreitern:

```
to Baum :g :d :v
if :v = 0 [stop] fd :g rt :d/2
Baum :g * 0.9 :d :v - 1 lt :d fd random 3
Baum :g * 0.9 :d :v - 1 rt :d/2 fd random 3 bk :g
end
```

Wird "Baum" mit dieser Prozedur aufgerufen, so bleibt nach Fertigstellung eine Ecke des Stammes offen. Sie wird später im Zusammenhang des ganzen Bildes mit **setpos** geschlossen. Die Prozedur "Bild" setzt nun einen Rahmen, rückt den Baum an die richtige Stelle, ruft die (hoffentlich noch im Arbeitsspeicher vorhandene - sonst neuzuschreibende) Kreisprozedur auf und beschriftet die entstandene Grafik mit **setcursor**. Hier kommen auch wieder die weiter oben beschriebenen Escape- oder **type word char 27 "** Folgen zur Geltung. (p/q = invers an/aus; r/u = Unterstreichen an/aus). Was dort nun geschrieben steht, bleibt der Phantasie überlassen. Dieser Text vielleicht mal als Beispiel für einen Briefkopf:

```
to Bild :g :d :v
type word char 27 "0
cs ct fs ht
Rahmen
pu setpos [-130 -250] pd Baum :g :d :v
setpos [-180 -250] pu setpos [-155 -245] pd fill
pu setpos [-250 150] pd Kreis 40 fill
ts setcursor [55 8] type [Garten & Landschaftsbau]
setcursor [55 10] type word char 27 "r type [Umweltpflege] type word char 27 "u
setcursor [55 12] type word char 27 "p type [E. Eiche GmbH] type word char 27 "q
fs
end
```

Theoretisch könnte man in dem entstehenden Bild mehrere Faktoren dem Zufallsgenerator Logo's überlassen. So etwa den Stand der Sonne. Allerdings müßten dann in die Prozedur noch Sicherungen eingebaut werden, damit es keine Überschneidungen gibt. Aber dies alles nur zur Anregung. Wurde alles richtig eingetippt, müßte nach dem Aufruf mit **Bild 60 50 5** (andere Zahlen ausprobieren!) die Grafik entstehen, die auf der folgenden Seite abgebildet ist.
Nach der Grafik nun noch einige Tips für Anwender, die sich nicht gern beim Programmieren den Kopf zerbrechen, und doch gern Computergrafik auf dem Papier hätten.



Man kann den Zeichencursor so lange etwas auf dem Papier verrichten lassen, bis ein Tastendruck ihn unterbricht. Diese Unterbrechung kann mit der Bedingung:

```
if keyp [stop]
```

erreicht werden. Dieses **keyp** fragt ab, ob eine Taste gedrückt wurde. Wenn die Meldung darüber positiv, also "true" ist, tritt die zweite Bedingung in Kraft und die Prozedur wird abgebrochen. Sieht eine Prozedur dann folgendermaßen aus:

```
to v
fd 2 if keyp [stop] v
end
```

läuft nach dem Aufruf mit **v** der Grafikcursor so lange vorwärts, bis eine Taste gedrückt wird. Da immer nur zwei Schritte gemacht werden, wird die Linie auch nicht zu schnell gezogen. So hat der Anwender reichlich Zeit, z.B. die Leertaste zu drücken um die Bewegung aufzuhalten und andere Prozeduren aufzurufen, die ähnlich erstellt werden. So etwa für den Rückwärtsschritt oder die Rechtsdrehung:

```
to r
rt 2 if keyp [stop] r
end
```

Unterteilt man vorher den Bildschirm mit **setsplit 2**, kann man trotz großen Grafikbereichs sogar noch seine Befehle auf dem Schirm sehen. Hat man sich an seine eigenen Befehle gewöhnt, wird auch das nicht mehr nötig sein und man kann mit der Turtle umgehen, als hätte man eine "Maus" zur Verfügung.
Noch besser wäre es, die Sache umzudrehen, nämlich die Turtle so lange in eine Richtung laufen zu lassen, wie eine bestimmte Taste gedrückt bleibt.
Mit **rc** erwartet Logo eine Eingabe von Tastatur, die mit **if**-Bedingung abgefragt werden kann.

```
if rc = "f [fd 5]
```

Nur dann, wenn die Taste [F] gedrückt wird, läuft der Cursor 5 Schritte nach vorn. Dies in eine Prozedur eingebaut, die sich selbst aufruft, würde, solange [F] gedrückt bleibt, die turtle nach vorn bewegen. Es ist nun keine Schwierigkeit, die entsprechenden Schritte für andere Richtungen einzubauen:

```
to z
setsplit 2
local "a make "a rc
if :a = "v [fd 5] (also vorwärts,z=zurück,r=rechtsherum etc)
if :a = "z [bk 5]
if :a = "r [rt 5]
if :a = "l [lt 5]
z
end
```

Nach dem Aufruf mit **cs ct z** , braucht der Anwender nur den Finger auf der Taste [V] zu halten, um die Turtle nach vorn zu bewegen. Ein längerer Wechsel zwischen [V] und [R] würde die Turtle in eine Rechtskurve zwingen. Da **setsplit 2** eingeschaltet wurde, stört es nicht, wenn mit der Stoptaste die Prozedur angehalten wird, um mal eben zwischen **pd** und **pe** etc. umzuschalten. Der Wiederaufruf mit "z gestaltet sich problemlos.
Wem auch dieser Tip keine Anregung geben kann, der sollte Logo wenigstens zum Nichtstun animieren. Mit:

```
repeat 975 []
```


wäre Logo etwa 3 Sekunden lang damit beschäftigt, nichts 975 mal zu wiederholen. Dies 19500 Mal zu versuchen, würde ihn somit etwa eine Minute lang beschäftigen, ohne daß etwas dabei herauskäme. Da Logo bei Werten über 32767 meint, daß ihm die Zahl zu groß sei, (Meldung: **number too big**) muß man schon **repeat**-Anweisungen verschachteln, will man ihn für längere Zeit zum Warten verdammen. Etwa fünf Minuten Wartezeit sähen dann so aus:

```
repeat 5 [ repeat 19500 [] ]
```

Vollkommen boshafte und militante Nicht-Logo-Anwender sind somit wenigstens in die Lage gesetzt, Logo und Joyce als Eieruhr zu mißbrauchen. Der Befehl:

```
type char 7
```

läßt Logo den Monitor einmal piepsen. Und so würde nach der Anweisung:

```
repeat 5 [ repeat 19500 [] ] repeat 50 [type char 7]
```

Logo den Joyce veranlassen, nach einer Frist von 5 Minuten einen ordentlichen Alarm zu veranstalten. Für ein Nickerchen läßt sich die Zeit selbstredend beliebig ausdehnen. Als Ausrede (und dem Leser als Anregung) für die Pieperei kann die Konstruktion eines Telefongebührenzählers vorgeschoben werden.
Eine komplette, mit Eingabeabfrage gestaltete Eieruhr könnte so aussehen:

```
to Eieruhr
pr [In wieviel Minuten moechten Sie von mir benachrichtigt werden ?]
local "a make "a rq
pr [Wie oft darf ich piepsen, um Sie zu alamieren ?]
local "b make "b rq
pr [Wenn sie jetzt eine Taste druecken, laeuft die Zeit]
label "warten
if not keyp [go "warten] ct cs
(pr [Ich bin jetzt damit beschaeftigt,\ ] :a [\ Minuten auf die Uhr zu achten])
repeat :a [repeat 19500 []] (pr [Die Zeit von\ ] :a [\ Minuten ist jetzt abgelaufen !!])
repeat :b [type char 7]
end
```

Das Vorherige setzt den Anwender jetzt auch in die Lage, eine "Analogo"-Uhr zu programmieren. Der Sekundenzeiger könnte mit Vor- und Rücklauf unter **px** Status seine Bahn ziehen, getaktet über eine **repeat[]**-Anweisung. Nach einer Drehung von 360 Grad könnte er den Minutenzeiger um 6 Grad nach rechts drehen, und den Stundenzeiger nach jeweils 12 * 360 Grad um 6 Grad weiterrücken.
Das Zifferblatt müßte nach dem Zeichnen mit **savepic** abgespeichert werden und vom Programm mit **loadpic** aufgerufen werden.
Wer sich bei der Ausarbeitung eines derartigen Programms - natürlich noch mit Weckfunktion - nicht den Kopf zerbrechen möchte, kann auf die Diskette zum Buch zurückgreifen. Es wird mit **load "wecker** in den Arbeitsspeicher geholt und durch Aufruf mit: **wecker** aktiviert. Nach der Initialisierung löscht dieses Programm einige seiner Teile, die nicht mehr benötigt werden, aus dem Arbeitsspeicher. Deshalb muß nach einem Abbruch des Programms vor dem Neustart die Prozedur immer wieder neu geladen werden.

Zuletzt

Das Bisherige mag einen kleinen Einstieg in die Welt der Turtle (und des Programmierens allgemein) gegeben haben.
An dieser Stelle soll jedoch explizit darauf hingewiesen sein, daß es sich wirklich nur um ein vorsichtiges Herantasten an die tatsächlichen Fähigkeiten dieser Sprache handelt. Der Bereich der Wort und Listenverarbeitung z.B. wurde in dieser Einführung überhaupt nicht angesprochen. Wollte man sämtliche Möglichkeiten, die die Sprache Logo bietet, erörtern, wäre das Resultat ein eigenständiges Werk und würde den Rahmen des Gesamtkonzepts dieses Buches sprengen.
Die Übersicht auf den nächsten Seiten über den kompletten Befehlssatz kann dem Anwender eine kleine Hilfe sein, sich weitere Bereiche selbst zu erarbeiten. Dort wurde eine ganze Reihe von nützlichen und erforderlichen Befehlen mit einem

Asterix (*) versehen. Diese Befehle sind in den meisten Schneiderhandbüchern zum Joyce nicht aufgeführt.
Wer durch die Anregungen animiert noch tiefer in die Materie Logo einsteigen will, dem sei das bereits im Vorwort erwähnte große Logo Buch zu CPC und Joyce von Sauer empfohlen. Es erschien bei Data Becker und ist über den DMV-Verlag zum Preis von 39,- DM zu bekommen. In diesem Buch wird auch gerade die Listenverarbeitung mit Logo als ein Schwerpunkt behandelt.

```
to Logo-Kapitel
if Inhalt = nicht Gedächtnis [noch einmal lesen]
end
```

Übersicht zum Logo-Befehlssatz

(Befehle mit Asteriks (*) sind im Schneider-Systemhandbuch nicht erklärt)

Befehl	Form	Erklärung
and	and "a<"b "c>"b	Gibt "true" aus, wenn die zwei gesetzten Ausdrücke beide wahr sind. Sonst ist die Ausgabe "false". In diesem Beispiel ist die Ausgabe "true", da die Reihe des Alphabets als aufsteigend gesehen wird. "And" darf nur in Präfix-Notation verwendet werden.
arctan	arctan 1	Gibt den Arcus-Tangens aus. Die Ausgabe bei arctan 1 wäre: 45. Die Angaben erfolgen in Grad.
ascii	ascii "g	Hat die Ausgabe des Ascii-Wertes eines Zeichens zur Folge. Hier wäre die Ausgabe 103. Von Listen wird immer nur das erste Element in Asciicode übersetzt.
bf	bf "Wort	But-first, also bis auf das erste Element einer Liste [in Klammern] oder eines Wortes, werden die anderen Elemente als Liste ausgegeben. In diesem Fall: ort
bk	bk 10; bk 5-2	Back dirigiert die Turtle zurück. Im ersten Beispiel um 10 Schritte. Die Schrittangabe kann auch in Form eines zu berechnenden Ausdrucks oder einer Variablen gegeben werden.
bl	bl "Wort	But-last, also bis auf das letzte Element einer Liste oder Wortes, werden alle Elemente ausgegeben. Hier: Wor (s. bf)
bye	bye	Logo wird verlassen und auf CP/M-Ebene zurückgekehrt. Die Prozeduren im Arbeitsspeicher gehen verloren.
catch*	catch "A [Prozed 2]	Mit throw und catch werden Sprünge über Prozeduren hinaus definiert. Throw "Anfang würde in diesem Beispiel einen Sprung zu der mit catch "A bezeichneten Stelle bewirken. Die zweite Angabe zu catch beinhaltet eine Liste mit ausführbaren Kommandos. Hier etwa eine Prozed 2 genannte Prozedur aufzurufen. Die unterschiedlichen Prozeduren, in denen throw und catch zu finden sind, müssen in einem übergeordneten Zusammenhang stehen. catch "error: Tritt während eines Programmablaufs ein Fehler auf, so kann dieser durch das eingefügte catch "error dahingehend abgefangen werden, daß Logo bei einem Eingabefehler automatisch den Rücksprung in die darauf folgende Zeile vornimmt.
changef		Dieser Befehl ist in Logo zwar vorgesehen, führt beim Joyce jedoch nur zu einer Fehlermeldung.
char	char 103	Gibt das Zeichen des Ascii-wertes aus. Hier wäre die Ausgabe "g
clean	clean	Bewirkt das Löschen des Grafikfensters.
co	co	(Continue) Nach Unterbrechung eines Programmablaufs durch pausing [F1] wird fortgefahren.
.contents	contents	Listet den Wortschatz des Arbeitsspeichers auf. Dabei sind auch Primitive wie paddle oder tones zu finden, die nicht auf dem Joyce implementiert sind.
copyoff	copyoff	Schaltet den Drucker aus.
copyon	copyon	Schaltet den Drucker ein. Sämtliche Aktionen, die über den Monitor laufen, werden protokolliert. Grafik kann so nicht erfasst werden.
cos	cos 60	Ergibt den Cosinus in Winkelgrad. Ausgabe wäre hier 0.5
count	count "Wort	Zählt die Buchstaben eines Wortes oder die Elemente einer Liste. Im zweiten Fall wäre count [a b c d e] die Ausgabe 5, im ersten Fall 4.

cs	cs	(Clear screen) Wie bei clean wird der Grafikschirm gelöscht. Zuätzlich wird der Grafikcursor in seine Ausgangsstellung gebracht, wobei die Spitze des Cursors nach oben weist. (s.a. seth 0)
ct	ct	(Clear Text) Der Textbereich wird gelöscht und der Textcuror in die obere linke Ecke seines Bereichs gesetzt.
cursor	cursor	Hat die genaue Stellung des Textcursors zur Ausgabe. Das erste Element der Liste bezeichnet die Spalte, das zweite die Zeile.
defaultd*	defaultd	Hat das Standardlaufwerk, mit dem Logo gerade arbeitet, zur Ausgabe.
define*	define "test :a	Kann innerhalb eines Programms ein neues Programm definieren, wobei die Meldung in diesem Beispiel: "test defined" unterbleiben würde. Hier würde jetzt in Test und unter dem Namen test die Prozedur a ablaufen.
dir	dir dir "m:	Ohne Laufwerksangabe werden die Logo-Files im Standartlaufwerk als Liste ausgegeben. Ansonsten wird auf das angesprochene Laufwerk zurückgegriffen.
dirpic	dirpic	Liefert die gespeicherten Bild-Dateien. (s. dir)
dot	dot [100 100]	Zeichnet einen Punkt an die durch Koordinaten bezeichnete Stelle. Koordinateneingabe muß eine Liste sein.
dotc*	dotc [100 100]	Wurde an der bezeichneten Koordinate ein Punkt gesetzt, hat dotc die Ausgabe 1. Wurde an diesem Koordinatenpunkt kein Zeichen vorgefunden, ist die Ausgabe 0.
ed	ed "test	Bereitet die Prozedur test zum Editieren vor und versetzt in den Editiermodus. Ed ohne Eingabe geht ohne weitere Angaben in den Editiermodus. Wird nach einer Fehlerausgabe ed eingetippt, wird die fehlerhafte Prozedur zum Editieren vorbereitet und der Cursor an die Stelle gerückt, in der der Fehler auftrat.
edall	edall	Bereitet alle im Arbeitsspeicher befindlichen Prozeduren zum Editieren vor und geht in den Editiermodus über.
edf*	edf "test	Lädt den File test von Diskette, bereitet ihn zum Editieren vor und geht in den Ed-Modus. Nach erfolgter Korrektur und Beendigung mit exit wird der korrigierte File wieder auf Diskette gespeichert.
emptyp*	emptyp :test	Primitive, die mit p enden, sind Prüfworte. In diesem Fall wird festgestellt, ob der Inhalt von test leer ist. Die Ausgabe ist true oder false. emptyp [] ergibt true.
end	end	Bezeichnet das Ende einer Prozedur.
equalp*	equalp "a "a	Prüft wie das =Zeichen ob zwei Dinge gleich sind. Ausgabe ist true oder false. Die Umkehrung ist not equalp.
er	er "Mwst	Löscht die Festlegung oder Prozedur Mwst.
erall	erall	Löscht den Inhalt des gesamten Arbeitsspeichers.
erasefile	erasefile "test	Löscht den File test ohne weitere Bestätigung von der Diskette.
erasepic*	erasepic "test	Löscht die Bilddatei test von Diskette.
ern	ern "a	Löscht den globalen Namen a.
error	error	Hat eine Liste zur Ausgabe, in der ein aufgetretener Fehler genannt wird. (s.a.catch)
fd	fd 10	Rückt den Grafikcursor hier um 10 Schritt nach vorn, wobei je nach Pencileinstellung Linien markiert werden. fd 0 malt an der Cursorposition einen Punkt.
fence	fence	Begrenzt das Grafikfeld, so daß beim Versuch dieses Feld zu überschreiten die Meldung: "Turtle out of bounds" ausgegeben wird.

fill*	fill	Füllt Flächen, in die der Grafikcursor unter pu hineingesetzt wird. Vor Aktivierung von fill wird der Status pd gesetzt. Fill kann während der Ausführung nur durch Abschalten des Computers gestopt werden. Sind Flächen nicht eindeutig eingegrenzt, wird der gesamte Monitor aufgefüllt!
first	pr first "test	Hätte in diesem Beispiel t zur Ausgabe. First gibt das erste Element eines Wortes oder einer Liste aus.
fput	show fput "a [b c]	Fput braucht zwei Eingaben, aus denen es eine Liste erzeugt. Die zweite Eingabe muß in Form einer Liste vorliegen. An diese Liste wird das erste Eingabeelement vorne angehängt. Hier wäre die Ausgabe: [a b c]
fs	fs	(Full screen) Stellt den gesamten Monitor als Grafikbereich zur Verfügung. Wird jedoch nach fs nicht cs (clear screen) eingegeben, bleibt wegen eines Systemfehlers in der Bildschirmmitte immer ein zweiter Grafikcursor stehen.
glist*	glist "ausmaß	Logo unterscheidet zwischen einem Ding, seiner Eigenschaft und dem Wert der Eigenschaft. Eigenschaften werden mit pprop (s.u.) zugewiesen. Hätte vor der Abfrage mit glist die Zuweisung pprop "test "ausmaß "groß stattgefunden, hätte also der Gegenstand test die Eigenschaft eines Ausmaßes, welches groß wäre, so würde die Abfrage in unserem Beispiel [test] ergeben. Glist stellt also fest, zu welchen Dingen eine bestimmte Eigenschaft gehört. (s.a. pps pprop remprop plist gprop)
go	go "vorne	Wurde innerhalb einer Prozedur mit label "vorne eine Markierung gesetzt, so wird nach einer Zeile mit go "vorne wieder in die label-Zeile gesprungen.
gprop*	gprop "test "ausmaß	(s. Beispiel glist) gprop gibt hier den Wert der Eigenschaft ausmaß des Dinges test aus. Ausgabe wäre hier also groß. Im Zusammenhang mit den Eigenschaftslisten tauchen Bezeichnungen auf, die groß geschrieben werden und denen ein Punkt vorangestellt wird. Diese Bezeichnungen sind in erster Linie für den Logo-Interpreter von Bedeutung. So zeigt: [.PRM und eine Zahl], daß die Eigenschaft zum Grundwortschatz von Logo gehört. Zur Probe: plist "fd hat zur Ausgabe: [.PRM 7020]. So zeigt .APV, daß es sich bei der Ausgabe um den Wert einer globalen Variablen handelt. Wurde mit make "test 10 definiert, so ergiebt: gprop "test ".APV die Antwort 10.
home	home	Dirigiert den Grafikcursor in seine Ausgangsstellung, wobei die Spitze nach oben zeigt.
ht	ht	(Hide turtle) Macht den Grafikcursor unsichtbar. Dadurch wird die Geschwindigkeit bei der Abarbeitung von Prozeduren wesentlich erhöht.
if	if :a<:b [stop]	If stellt eine Bedingung. In diesem Fall die Bedingung, daß a kleiner als b ist. Trifft diese Bedingung zu,(if arbeitet mit true oder false) wird die darauf folgende Liste zur Ausführung gebracht. In diesem Fall würde eine gesamte Prozedur gestopt. Trifft die Bedingung nicht zu, wird zur nächsten Anweisung übergegangen. if keyp [stop] würde dann eine Prozedur abbrechen, wenn die Tastatur betätigt wird.
int	int 3.141	Gibt den ganzzahligen Wert einer Zahl aus. Hier wäre die Ausgabe 3
item	item 2 :tf	Mit item wird aus einer Liste ein Element herausgefiltert. Hier wäre es aus der Liste der Turtle-facts das zweite Element.

keyp*	keyp	Hier handelt es sich um ein Prüfwort, welches true ausgibt, wenn eine Taste gedrückt wird. Es ist besonders gut für von außen steuerbare Abbruchbedingungen von Prozeduren geeignet. (s. if)
label	label "vorne	Label markiert hier eine Sprungadresse für go "vorne. Die Sprünge finden innerhalb einer Prozedur statt.
last	pr last "vorne	Last hat im Gegensatz zu first das letzte Element einer Liste oder eines Wortes zur Ausgabe. In diesem Fall wäre es "e.
lc	lc "MwSt	Gibt das eingegebene Wort lediglich in kleinen Buchstaben wieder. Hier mwst.
list	(list "a "b "c)	Die angeführten Elemente werden zu einer Liste zusammengefasst und als solche ausgegeben. Hier [a b c]
listp*	listp :a	Untersucht wird hier, ob es sich bei dem Wert von a um eine Liste handelt. Wie bei den anderen Prüfworten ist die Ausgabe true oder false.
load	load "test	Eine Datei mit dem Namen test wird nach diesem Befehl in den Arbeitsspeicher geladen.
loadpic	loadpic "strip	Ein mit savepic abgespeichertes Bild wird mit loadpic wieder in den Arbeitsspeicher geladen. Hier handelt es sich um ein Bild mit dem Namen strip.
local	local "a	Die Variable a wird zur lokalen Größe erklärt, die ihren Wert nur für einen Durchlauf der Prozedur erhält, in der sie auftritt.
lput	show lput "a [b c]	Es handelt sich um die gegenteilige Funktion von fput. Hier wird an das Ende einer Liste ein Element angehängt und mit dem Resultat eine neue Liste eröffnet. Hier [b c a]
lt	lt 90	Dreht den Grafikcursor um 90 Grad nach links.
make	make "a 100	Make weist einer Variablen einen Wert zu. Hier erhält a den Wert 100.
memberp*	memberp "o "Wort	Auch hier verrät das p am Ende von memberp das Prüfwort. Abgefragt wird, ob das zuerst genannte Element in dem folgenden Wort oder der Liste enthalten ist. Zutreffend wird true ausgegeben, sonst false.
namep*	namep "a	Es wird nachgeprüft, ob in diesem Beispiel a im Arbeitsspeicher belegt ist. Ausgabe ist true oder false.
nodes	nodes	Gibt den zur Verfügung stehenden Platz im Arbeitsspeicher aus. Vor der Abfrage sollte der Speicher mit recycle behandelt werden.
not	not 2 < 1	Gibt true aus, wenn eine Aussage - wie in diesem Fall - falsch ist.
notrace*	notrace	Schaltet den mit trace erreichten Überwachungszustand ab.
nowatch*	nowatch	Schaltet den mit watch erreichten Überwachungszustand ab.
numberp*	numberp :a	Das p am Ende verrät hier das Prüfwort. Es wird nachgeprüft ob es sich beim Wert von a um eine Zahl handelt. Die Ausgabe ist true oder false. Mit numberp können Tastatureingaben, die z.B. nach rc eine Zahl erwarten, mit if-Bedingung abgefangen werden, falls sich der Anwender vertan hat und z.B. einen Buchstaben eingetippt hat.
op	op proz :a	Der Wert von a wird mit op (output) ausgegeben. In diesem Fall würde op den Wert von a an eine andere Prozedur mit dem Namen proz übergeben, die dann mit a weiterarbeitet.
or	or (1<2) (2<1)	Wenn mindestens eine der zwei Bedingungen, die hinter or stehen, wahr ist, ist die Ausgabe true. Trifft keine der Aussagen zu, ist die Ausgabe false.
pause	pause	Mit pause wird eine Prozedur unterbrochen (nicht abgebrochen). Die Prozedur kann in diesem Zustand auch verändert werden und anschließend mit co fortgesetzt werden. Bei richtiger Tastaturbelegung bewirkt [F1] auch ein pausing.

pd	pd	(Pen down) Der Zeichencursor zieht bei Bewegung eine Linie hinter sich her.
pe	pe	(Pen eraser) Der Zeichencursor löscht Linien, über die er hinweggeführt wird.
piece*	piece 2 4 tf	Aus der Liste z.B. der tf (turtle facts) würde in diesem Fall das 2.bis 4.Element aussortiert und als Liste ausgegeben.
plist*	plist "a	Wurde mit make "a 1 gesetzt, so ergibt die Frage nach a mit plist : [.APV 1] (vergl. glist). Mit plist wird also die Eigenschaft eines Dinges erfragt. .APV = associated property value ist die Bezeichnung für Eigenschaft.
po	po "test	Bringt den Programmtext der Prozedur "test" aus dem Arbeitsspeicher in das Textfenster.
poall	poall	Bringt alle Programmtexte der im Arbeitsspeicher befindlichen Prozeduren ins Textfenster. Ist dieses vollgeschrieben, wartet Logo mit weiteren Eingaben, bis die Returntaste gedrückt wurde.
pons	pons	(print out names) Mit pons werden alle Namen und ihre zugehörigen Werte ausgegeben.
pops	pops	(print out procedures) Alle Prozedurtexte und Werte der Namen werden ausgegeben.
pots	pots	(print out titles) Bringt die selbstdefinierten Prozedurnamen nebst den erwarteten Eingaben.
pprop*	pprop "test "groß pprop "test "gut	(s. glist, plist) Weist einem Objekt eine Eigenschaft zu, die mit plist wieder abgefragt werden kann. plist "test ergäbe [gut groß]
pps*	pps	Die Abfrage mit pps ergibt alle Namen des Arbeitsspeichers mit den ihnen zugewiesenen Eigenschaften.
pr	pr 100 - 50	(Print) Das Ergebnis wird als Ausgabe an den Monitor gegeben.
pu	pu	(pen up) In diesem Zustand bewirkt die Bewegung des Grafikcursors nichts auf dem Monitor.
px*	px	(pen xor) Trifft der Grafikcursor bei seiner Bewegung auf gesetzte Punkte, werden diese gelöscht. Sind keine Punkte vorhanden, werden welche gesetzt.
quotient	quotient 10 4	Als Ausgabe hat quotient das ganzzahlige Ergebnis aus einem Bruche. Hier 10/4, also 2
random	random 6	Ermittelt wird hier eine Zufallszahl. Die Ausgabe bewegt sich in diesem Fall zwischen 0 und 6.
rc	rc	(read character) Nach rc erwartet Logo eine Eingabe von Tastatur, die nicht auf den Monitor ausgegeben wird, die kein Return erwartet und sofort programmintern verwertet wird.
recycle	recycle	Der Arbeitsspeicher wird von unnötig belegten nodes, die Logo für die Organisation gebraucht hat, gereinigt. Durch recycle werden oft erhebliche Speicherplatzkapazitäten frei. Es sollte deshalb vor größeren Operationen durchgeführt werden.
remainder	remainder 10 4	Hier wird der Rest der Ganzzahldivision ausgegeben. In diesem Fall: 2.
remprop*	remprop "test "a	Mit remprop wird die Eigenschaft eines Objektes wieder entfernt. Hier wird dem Objekt "test die Eigenschaft "a genommen.
repeat	repeat 9 [fd 5]	Es werden zwei Eingaben erwartet. Die erste sagt repeat, wie oft es etwas tun soll, die zweite Eingabe - eine Liste - sagt, was es tun soll. Repeat 19500 [] würde bewirken, daß die leere Liste 19500 Mal durchgeführt würde, was weiter nichts bewirken würde, als das der Computer für ca. eine Minute in einen Wartezustand versetzt würde.
rerandom*	rerandom	Bei jedem Aufruf mit random wird aus der Liste des Zufallsgenerators eine vorherbestimmte Zahl geholt. Diese Liste wird mit rerandom wieder auf ihren ersten Wert gesetzt.

rl	rl	(read list) Es wird eine Liste als Eingabe erwartet, die erst nach Drücken von Return verarbeitet wird. Das Ergebnis von rl kann gut weiterverarbeitet werden. count rl würde z.B. die Elemente einer Eingabe zählen.
round	round 3.14	Es wird auf ganze Zahlwerte gerundet. Ausgabe wäre hier 3.
rq	rq	Ähnlich wie rl, nur daß hier als Eingabe nur ein Element erwartet wird. Ebenfalls erst nach Return.
rt	rt 90	Dreht den Zeichencursor ausgehend von seiner momentanen Stellung (hier um 90 Grad) nach rechts.
run	run [a b c]	Run führt eine Befehlsliste aus. Wären a b c Prozeduren, würden sie aktiviert.
save	save "test	Sichert die im Arbeitsspeicher befindlichen und selbstgeschriebenen Prozeduren unter dem Namen Test auf Diskette des aktuellen Laufwerks.
savepic	savepic "test	Sichert eine Grafik als Bilddatei (extension .pic) unter dem Namen test auf dem Standartlaufwerk. Der Vorteil der Bildspeicherung besteht darin, daß die Grafiken mit loadpic "name wesentlich schneller auf den Monitor zu bringen sind. Außerdem sind Hardwareerweiterungen (scanner) und Software (desktop-publisher) auf dem Markt, die mit diesen pic-Dateien kompatibel sind.
se	se :a :b	Mit se können verschiedene Elemente zu einer Liste zusammengefasst werden. Will man z.B. bei setpos mit den Werten von :a und :b arbeiten, müssen diese, da setpos eine Liste mit zwei Elementen erwartet, mit se zusammengefasst werden.
setcursor	setcursor [45 15]	Hier wird der Textcursor in Spalte 45 und Zeile 15 (Bildschirmmitte) gesetzt.
setd*	setd "m:	Mit setd kann das Standartlaufwerk gewählt werden. In dieser Eingabe würde M: als solches angesprochen werden.
seth	seth 90	Von einer absoluten Stellung ausgehend (0=oben) wird die Turtle um z:B. 90 Grad gedreht. Nach seth 90 würde sie immer nach rechts weisen.
setpc*	setpc 0	Setpc kennt nur zwei Stellungen, 0 oder 1. Der Effekt bei der Grafikprogrammierung ist derselbe wie bei pe (0) oder pd (1).
setpos	setpos [0 0]	Setzt den Grafikcursor auf die durch die Koordinatenpunkte bestimmte Stelle, hier in den Bildmittelpunkt. Als Eingabe wird eine Liste erwartet.
setscrunch	setscrunch 0.5	Damit wird das Achsenverhältnis der Koordinaten geändert. In dem Beispiel wurde von der Standarteinstellung abgewichen.. Maximal kann das Verhältnis im Maßstab 1:10/10:1 (0.46875 Standard) geändert werden.
setsplit	setsplit 2	Damit wird der Monitor in einen Grafik und einen Textbereich geteilt. Die Zahl hinter setsplit gibt die Größe des Textschirms in Zeilen an.
setx	setx 10	Die Turtle wird parallel zur x-Achse des Koordinatensystems bewegt und auf den Punkt 10 dieser Achse gesetzt.
sety	sety 10	Entsprechend setx. Nach Absolvierung dieser Beispiele stände die Turtle auf dem Punkt, der auch durch setpos [10 10] erreicht worden wäre.
sf	sf	(screen facts) Berichtet über den Zustand des Bildschirms. Die Elemente der Liste bedeuten der Reihe nach: Hintergrund des Grafikfensters (0/1); Fensteraufteilung (ts ss oder fs); Anzahl der Textzeilen bei ss oder setsplit; Art der Bildschirmbegrenzung (fence window oder wrap); letztlich das Achsenverhältnis (Standart 0.46875).
show	show [a b c]	Zum Kenntlich machen von Listen. Die Klammern werden mit ausgegeben. Hier: [a b c]

shuffle	shuffle [a b c d]	Wirft Elemente einer Liste nach Zufallskriterien durcheinander, und gibt sie als Liste aus. Hier z.B. [b d a c]
sin	sin 30	Ausgabe ist der Sinuswert im Gradmaß des angegebenen Winkels Hier Ausgabe: 0.5
ss	ss	(Standart split) Grafik- und Textbereich des Monitors werden nach dem Standart aufgeteilt.
st	st	(show turtle) Macht den mit ht versteckten Grafikcursor wieder sichtbar.
stop	stop	Bricht Prozeduren ab.
text	text "test	Hier wird der Text der Prozedur "test" ausgegeben.
tf	tf	(turtle facts) Liefert Information über die Zustände der Turtle. 1.u.2. Position im Koordinatensystem, 3. Richtung der Turtle (seth), 4. zeigt, ob pd, pe, px oder pu aktiviert sind. 5.: Farbe des Zeichenstifts (Joyce 0/1) 6.: Zeigt ob Turtle sichtbar ist (true oder false; st/ht)
thing	thing "a	Hat den Wert der Liste a zur Ausgabe.
throw	throw "anfang	(s. catch) Hier würde die Prozedur an die Stelle einer untergeordneten Prozedur springen, die mit catch "anfang bezeichnet wurde. Eine Prozedur kann auch wie bei einem Abbruch mit der Stoptaste verlassen werden, ohne daß die Stopped-Meldung auf dem Monitor erscheint. Dies wird durch die Vokabel "TOPLEVEL (groß schreiben!) erreicht. throw "TOPLEVEL kehrt in den Direktmodus zurück.
to	to "test	To kennzeichnet den Namen einer Prozedur. Wird im Direktmodus to "test eingegeben, stellt Logo ein eingeschränktes Editierfeld zur Verfügung, in dem nicht zeilenweise gesprungen werden kann.
towards	towards [10 10]	Richtet die Spitze des Grafikcursors auf die in der nachfolgenden Liste angegebenen Koordinatenpunkte.
trace*	trace	Es wird ein spezieller Überwachungszustand eingeschaltet, der dem Anwender zeigt, an welcher Stelle sich eine ablaufende Prozedur befindet und die Werte der Variablen angibt (s. notrace).
ts	ts	(text screen) Der gesamte Monitor wird als Textbereich genutzt
type	type [a b c]	Stellt a b c auf dem Monitor dar, ohne daß ein Zeilenumbruch erfolgt. Die Klammern werden weggelassen.
uc	uc "mwst	(upper case) Die Ausgabe des nachstehenden Wortes erfolgt in Großbuchstaben. Hier: MWST
watch*	watch	Überwachungsmodus wie trace. Es werden jedoch die Verschachtelungstiefe und momentan ausgeführte Programmzeile angezeigt. Erst nach Druck auf Return wird der nächste Befehl des beobachteten Programms ausgeführt. (s. nowatch)
where* (pr memberp "b "abc where)		Memberp würde in diesem Beispiel feststellen, ob das Element b in abc vorhanden ist. Bei positiver Beantwortung gibt where jetzt den Standort von b in abc - also 2 - aus.
window	window	Der Grafikcursor kann in diesem Modus über den sichtbaren Bereich hinaus gesteuert werden.
word	pr word "com "puter	Ähnlich wie "se" eine Liste erstellt, erzeugt word aus Elementen ein Wort. Hier "computer
wordp*	wordp :a	Als Prüfwort fragt wordp nach, ob es sich bei der Eingabe um ein Wort handelt. Ausgabe ist true oder false.
wrap	wrap	Verläßt in diesem Modus der Grafikcursor den sichtbaren Bereich des Monitors, erscheint er am gegenüberliegenden Punkt des Bildschirms.

B A S I C

Vorweg

Eine derartig ausführliche Einführung in BASIC, wie sie im vorherigen Kapitel über Logo gegeben wurde, erschien nicht nötig, da das Benutzer-Handbuch schon über viele Aspekte der Programmierung Auskunft gibt. Einiges wird jedoch selbst dem fortgeschrittenen Einsteiger nicht unbedingt klar, so daß auch grundsätzliche Dinge wie Speichern, Laden oder Aufruf von Programmen etc. bei der Arbeit mit BASIC vorweg kurz anzusprechen sind.
Hingegen ist es auf jeden Fall angebracht, die einzelnen BASIC-Befehle noch einmal aufzuführen und mit einem kleinen Beispielprogramm zu versehen, da das Benutzer-Handbuch den Anwender oft genug im unklaren darüber läßt, wie denn Befehle zu handhaben sind. Außerdem kommen bei vielen Anweisungen eine Unzahl von Einzelaspekten und Feinheiten hinzu, die im Benutzer-Handbuch gar keine Erwähnung finden oder nur sehr verwirrend dargestellt sind.
Die im folgenden alphabetisch angeführten Befehle sollen dem Anwender, der sich schon ein wenig in BASIC auskennt, eine zusätzliche Hilfe bei der Programmierung sein und dem Neuling durch die Programmbeispiele Strukturen des BASIC verdeutlichen.
Ebenso wie im Logo-Teil finden sich die Listings zu den Beispielen ready to run auf der zum Buch erhältlichen Diskette.

In vielen Fällen weiß der Programmierer bei seiner Arbeit zwar, was er erreichen will, dafür aber nicht, welcher Befehl ihn seinem Ziel näherbringt. Bei solchen Problemen soll die Zusammenstellung der BASIC-Befehle nach Einsatzgebieten weiterhelfen.
Ist das Benutzer-Handbuch in punkto Erklärung zu BASIC-Befehlen schon recht schwach, so ist die Dokumentation zu Jetsam im Kapitel 8 des Benutzer-Handbuches kaum noch eine solche zu nennen. Hier wird auf kaum 20 Seiten versucht, dem Anwender das Prinzip der Jetsamverwaltung zu erläutern.

Um diesem Manko entgegenzutreten, erwies es sich als nötig, ein wenig genauer die Programmierung mit Jetsam an beispielhaften Programmzeilen zu erklären. Außerdem wurde auf die zum Buch erhältliche Diskette das Programm "Generjet" mit aufgenommen, das dem Anwender Jetsamstrukturen erklärt und außerdem bei der Erstellung von Jetsamprogrammen interaktiv weiterhilft. Dieses Programm stellt ein wertvolles "tool" dar, ist aber leider so umfangreich, daß das Listing im Buch zu viel Platz in Anspruch genommen hätte. (vgl. dazu auch die Anleitung im Text).

Allgemeine Einführung

in BASIC

Im folgenden soll eine Einführung in die Programmiersprache BASIC gegeben werden, die sich an den absoluten Laien richtet, der zum ersten Mal an einem Computer sitzt und keine Vorkenntnisse besitzt.

Um die Programmiersprache BASIC benutzen zu können, muß sie erst einmal in den Computer geladen werden. Dies geschieht in drei Schritten:

1. Betriebssystem CP/M Plus laden
2. BASIC-Diskette einlegen
3. BASIC laden

Zu 1.: Das Laden des Betriebssystems geschieht beim JOYCE durch Anschalten des Rechners und Einlegen einer Systemdiskette in Laufwerk A:, wobei die Seite mit dem Betriebssystem zum Bildschirm zeigen muß. Bei den dem JOYCE beigefügten Disketten befindet es sich auf Seite 2.

Zu 2.: Jetzt muß die BASIC-Diskette ins Laufwerk eingelegt werden. Wenn Sie gerade das Betriebssystem geladen haben, können Sie diesen Schritt überspringen, weil sich BASIC zusammen mit dem System auf einer Diskettenseite befindet.

Zu 3.: Das eigentliche Programm wird durch Eingabe von **BASIC** und einem abschließenden Drücken der [RETURN]- oder [ENTER]-Taste geladen. Der JOYCE sucht nun das Programm auf der Diskette. Falls vorhanden, lädt er es in den Speicher und startet es.

Wenn diese drei Schritte erfolgreich absolviert wurden, müßte jetzt die Einschaltmeldung des Mallard-80 BASIC auf dem Bildschirm erscheinen. Zum Schluß steht dann:

```
OK
■
```

Wenn Sie an dieser Stelle angelangt sind, befinden Sie sich im Direktmodus von BASIC. Das heißt, daß Sie nun durch das ■ aufgefordert werden, BASIC irgendwelche Kommandos zu geben, durch die BASIC weiß, was es jetzt zu tun hat.

An dieser Stelle fangen wir am besten mit einem Miniprogramm an:

```
10 PRINT "Dies ist mein erstes Programm."
20 END
```

Sobald Sie eine Zeile fertig eingegeben haben, muß sie mit einem abschließenden Drücken der Taste [RETURN] oder [ENTER] im Speicher fixiert werden.

An diesem Beispiel sieht man, daß jede Zeile, die BASIC abarbeiten soll, mit einer Zeilennummer anfängt. Danach schließen sich dann die Kommandos an. In unserem Beispiel ist es das Kommando **PRINT**. Durch **PRINT** wird BASIC signalisiert, das dem Kommando Folgende auf dem Bildschirm auszugeben. Auf dem Bildschirm erscheint also:

```
Dies ist mein erstes Programm.
```

In Zeile 20 steht das Kommando **END**. Es sagt BASIC, daß das Programm an dieser Stelle zu Ende ist. Nach dessen Ausführung springt BASIC wieder in den Direktmodus.

Um jetzt das Programm zu starten und dessen Wirkung zu beobachten, müssen Sie **RUN** eingeben und mit [RETURN] abschließen. Das Programm wird gestartet und auf dem Bildschirm erscheint:

```
Dies ist mein erstes Programm.
Ok
■
```

Das zweite Beispielprogramm soll nun etwas komplexer als das erste ausfallen: als Neuerung kommen Variablen.


```
10 a=10
20 b=15
30 c=a*b
40 PRINT "Ergebnis: ";c
50 END
```

In Zeile 10 wird der Variablen **a** der Wert 10, in Zeile 20 der Variablen **b** der Wert 15 zugewiesen. Danach wird in Zeile 30 die Multiplikation von **a** und **b** ausgeführt und das Ergebnis der Variablen **c** zugewiesen. Zum Schluß wird das Ergebnis auf dem Bildschirm ausgegeben. Das Semikolon bewirkt dabei, daß der Wert der Variablen **c** von dem Text nicht durch zusätzliche Leerzeichen getrennt wird.

Wenn das Programm korrekt eingegeben und mit **RUN** gestartet wurde, müßte auf dem Bildschirm

```
Ergebnis:  150
Ok
■
```

erscheinen. Ist dies nicht der Fall, so haben Sie sich in irgendeiner Zeile vertippt. Um jetzt nicht die ganze Zeile noch einmal von vorne neu eingeben zu müssen, stellt BASIC das Kommando **EDIT** zur Verfügung. Sollten Sie sich also zum Beispiel in der Zeile 40 vertippt haben, so müssen Sie

```
EDIT 40
```

eingeben, gefolgt von einem [RETURN]. Auf dem Bildschirm erscheint dann die entsprechende Zeile, in der Sie mit den Cursortasten beliebig hin- und herfahren und die entsprechende Stelle ändern können. Um das Ändern etwas zu erleichtern, gibt es einen Überschreibmodus, der durch Drücken der [+]-Taste (links neben Space) eingeschaltet und durch nochmaliges Drücken wieder abgeschaltet werden kann. Beim Überschreibmodus können die alten Zeichen von neuen überschrieben werden, d.h. der nachfolgende Text wird nicht nach rechts verschoben. Wenn die Zeile korrigiert wurde, muß man wieder [RETURN] betätigen, um sie in der neuen Form abzuspeichern. Um sich davon überzeugen zu können, daß die Zeile im Programm auch wirklich geändert wurde, verwendet man den Befehl **LIST**. Er bewirkt, daß das komplette Programm auf dem Bildsohirm ausgegeben wird. Zusätzlich kann man sich jedoch

auch nur einen Ausschnitt des Programms anschauen. Dazu teilt man **LIST** noch einen Zeilenbereich mit. Um zum Beispiel die Zeilen 10 bis 30 aufzulisten, wird

```
LIST 10-30
```

(wieder + [RETURN]) eingegeben. Weitere Möglichkeiten des **LIST**-Kommandos entnehmen Sie bitte der ausführlichen Erläuterung im BASIC-Kommando-Teil dieses Buches, in dem auch alle anderen in der BASIC-Einführung verwendeten Befehle erklärt werden.

Unser bisheriges Programm läßt jedoch noch einigen Komfort vermissen. Es wäre zum Beispiel sinnvoll, daß man die beiden Werte, mit denen das Programm rechnen soll, eingeben kann. Dazu müssen wir jedoch ein weiteres BASIC-Kommando kennenlernen: **INPUT**. Damit wird es möglich, Werte und Texte Variablen zuzuweisen. Wenn das Beispielprogramm entsprechend geändert wird, sieht es folgendermaßen aus:

```
10 INPUT "Wert a: ",a
20 INPUT "Wert b: ",b
30 c=a*b
40 PRINT "Ergebnis: ";c
50 END
```

Die beiden **INPUT**-Anweisungen in Zeile 10 und Zeile 20 bewirken, daß zuerst auf dem Bildschirm der Text zwischen den Anführungszeichen erscheint und man hinter dem Text die Werte eingeben kann (eine ausführlichere Beschreibung mit weiteren Optionen entnehmen Sie bitte dem Kommando-Teil dieses Buches). Wenn man das Programm mit **RUN** (+[RETURN]) startet, könnte ein Testlauf folgendermaßen aussehen:

```
Wert a: -6
Wert b: 10
Ergebnis: -60
Ok
■
```

Der nächste wichtige Schritt im Leben eines Programmierers ist, seine Arbeit dauerhaft auf einem Datenträger zu sichern. Zu diesem Zweck wird von BASIC der Befehl **SAVE** bereitgestellt. Benutzt man ihn, so wird das komplette Programm in der derzeitigen Form mit einem bestimmten Namen auf der Diskette abgespeichert. Wenn Sie das Programm zum

Beispiel unter dem Dateinamen "RECHNE" speichern wollen, so müssen Sie nur

```
SAVE "RECHNE
```

(+ [RETURN]) eingeben. Das Programm wird jetzt unter dem Dateinamen "RECHNE.BAS" auf Diskette geschrieben. Das ".BAS" besagt, daß es sich bei dieser Datei um ein BASIC-Programm handelt.

Genauso wichtig wie das Speichern von Programmen ist das spätere Laden derselben. Es geschieht mit dem Kommando **LOAD**. Um "RECHNE.BAS" wieder zu laden, genügt ein einfaches

```
LOAD "RECHNE
```

Das ".BAS" braucht man nicht mit anzugeben, da es von BASIC selbständig ergänzt wird.

Das Programm ist nun wieder zur weiteren Bearbeitung in den Speicher geladen worden.

Eine Erweiterung des Programms wird im folgenden dargestellt: es soll möglich sein, die Verknüpfungsart der Zahlen frei wählen zu können.

```
10 INPUT "Wert a: ",a
20 INPUT "Wert b: ",b
30 INPUT "Rechenart (+ , - , * , /): ",r$
40 IF r$="+" THEN c=a+b
50 IF r$="-" THEN c=a-b
60 IF r$="*" THEN c=a*b
70 IF r$="/" THEN c=a/b
80 PRINT "Ergebnis: ";c
90 END
```

Man sieht, daß das Programm um die Zeilen 30-70 ergänzt wurde. Hauptbestandteil sind dabei die **IF**-Zeilen, in denen die eingegebene Rechenart geprüft und ausgeführt wird. In der Variablen **r$** steht das Symbol der auszuführenden Rechenart, das man über die in Zeile 30 stehende **INPUT**-Anweisung eingegeben hat. Das Symbol wird mit der Bedingung ("+","-", "*", "/") verglichen. Falls diese wahr ist, wird der Variablen **c** das Ergebnis aus der Verknüpfung von **a** und **b** zugewiesen.

In dem dritten Beispiel soll nun das Berechnungsproblem der Primzahlen abgehandelt werden. Grundlage ist die Eigenschaft einer Primzahl: eine Primzahl hat keinen Teiler außer 1 und ihrer selbst. Das heißt umgekehrt, daß alle Vielfachen einer Primzahl keine Primzahlen sein können ! Beispiel: man lege sich eine Tabelle aller Zahlen ab 2 (1 ist keine Primzahl) an. Dann streiche man alle Vielfachen von 2 aus (2 bleibt stehen) und prüfe ob die Zahl 3 ausgestrichen ist. Sie ist es nicht und ist daher eine Primzahl. Als nächstes streiche man alle Vielfachen von 3 aus. Dann prüfe man die Zahl 4. Da sie ausgestrichen ist, ist sie keine Primzahl. Nun wird die Zahl 5 geprüft... usw.

```
10 DEFINT z
20 grenze=1000
30 DIM z(grenze)
40 i=2
50 IF z(i)=0 THEN PRINT i ELSE GOTO 90
60 FOR u=i TO grenze STEP i
70     z(u)=1
80 NEXT
90 i=i+1
100 IF i<=grenze THEN 50
110 END
```

In Zeile 10 wird die Feldvariable **z** als Integervariable (ganzzahlige Variable) definiert und in Zeile 20 die obere Primzahlberechnungsgrenze auf 1000 (maximal 15675) gesetzt. In Zeile 30 wird die Feldvariable **z** als ein Feld von 0 bis 1000 definiert. In Zeile 40 beginnt die eigentliche Berechnung, angefangen bei der ersten Primzahl 2. Zeile 50 prüft, ob die Variable als ein Vielfaches einer anderen Zahl definiert ist. Ist sie es, werden in den Zeilen 60-80 alle Vielfachen der Zahl als Vielfache markiert, indem die entsprechende Feldvariable auf eins gesetzt wird. In Zeile 90 wird die zu prüfende Zahl um eins erhöht und anschließend geprüft, ob die obere Grenze überschritten wurde und daher in Zeile 110 das Programm beendet werden muß.

Dieses Verfahren hat den Vorteil, das es relativ schnell arbeitet. Aber auch der Nachteil liegt klar auf der Hand: man hat eine obere Grenze, bis zu der man Primzahlen

berechnen kann, weil der Speicherbereich für die Feldvariable **z** zu knapp wird.

Im vierten Beispiel treten die genannten Nachteile nicht auf, allerdings dauert es im Gegensatz zum vorherigen Beispiel mit steigender Primzahl immer länger, bis die jeweils nächste gefunden wird.

Das Prinzip dieses Verfahrens ist, daß geprüft wird, ob bis zur Hälfte der Zahl ein ganzzahliger Teiler vorhanden ist. Wenn es keinen Teiler gibt, muß es sich um eine Primzahl handeln.

```
10 i=2
20 FOR u=2 TO i/2
30     IF i/u = INT(i/u) THEN 60
40 NEXT
50 PRINT i
60 i=i+1
70 GOTO 20
```

Da durch den Sprung in Zeile 70 zum Anfang des Programms eine Endlosschleife realisiert ist, kann das Programm irgendwann durch Betätigen des [STOP]-Taste oder durch [ALT]+[C] abgebrochen werden.

Wenn Sie die Primzahlen in den beiden letzten Beispielen lieber hintereinander statt untereinander auf den Bildschirm bringen möchten, müssen Sie jeweils an das Ende der PRINT-Anweisung ein Semikolon setzen:

```
50 IF z(i)=0 THEN PRINT i; ELSE GOTO 90
```

bzw.

```
50 PRINT i;
```

Zum Schluß dieser Kurz-Einführung soll noch gezeigt werden, wie man Daten auf dem Drucker ausgibt, wenn man seine Programme bzw. Primzahlen gerne dauerhaft auf Papier haben möchte.

Das Ausgeben eines Programms ist denkbar einfach: ähnlich dem Auflisten auf dem Bildschirm mit dem **LIST**-Kommando kann das Programm mit **LLIST** auf dem Drucker ausgegeben werden. Das **LLIST**-Kommando ist genauso wie das **LIST**-Kommando zu

verwenden, was heißt, daß ohne Einschränkungen Zusätze wie die Wahl eines Zeilenbereiches verwendet werden können.

Auch das **PRINT**-Kommando wird einfach mit einem **L** davor, also **LPRINT**, auf den Drucker umgelenkt. Außerdem kann es wie das **LLIST**-Kommando ohne Einschränkungen gegenüber dem einfachen **PRINT** verwendet werden.

```
LLIST
```

```
LLIST 10-30
```

```
50 IF z(i)=0 THEN LPRINT i; ELSE GOTO 90
```

bzw.

```
50 LPRINT i;
```


Zusammenfassung aller BASIC-Befehle nach Funktionsgruppen

Abfrage

IF	Bedingung prüfen

Bildschirmausgabe

POS	aktuelle Position des Cursors
PRINT	Text ausgeben
SPC	Leerstellen ausgeben
TAB	auf Tabulatoren springen
WIDTH	Bildschirmzeilenbreite bestimmen
WRITE	Text ausgeben

Diskettenarbeit

OPEN	Datei eröffnen
CLOSE	Datei schließen
DIR	Directory ausgeben
FILES	Directory ausgeben
DISPLAY	Datei auflisten
TYPE	Datei auflisten
EOF	prüfen, ob Dateiende erreicht
ERA	Datei löschen
KILL	Datei löschen
FIND$	Datei finden
INPUT #	Satz aus Datei lesen
INPUT$	n Zeichen aus Datei lesen
LINE INPUT	Datei zeilenweise auslesen
PRINT #	in Datei schreiben
WRITE #	in Datei schreiben
LOAD	Programm laden
RUN	Programm laden und starten
SAVE	Programm abspeichern
LOF	Dateilänge ermitteln
NAME	Datei umbenennen
REN	Datei umbenennen
OPTION FILES	Laufwerk und Benutzernummer ändern
RESET	Diskettenlaufwerk zurücksetzen

Drucker

LLIST	Programm auf Drucker ausgeben
LPOS	aktuelle Position des Druckkopfes
LPRINT	Text auf Drucker ausgeben
WIDTH LPRINT	Druckzeilenlänge bestimmen

Fehlerbehandlung

ERL	Fehlerzeile
ERR	Fehlernummer
ERROR	Fehler erzeugen
ON ERROR GOTO	Fehlerbehandlungsroutine anspringen
RESUME	normales Programm nach Fehler fortsetzen
TRON	Programmzeilenprotokoll einschalten
TROFF	Programmzeilenprotokoll ausschalten

Formatierte Zahlenausgabe

DEC$	Zahl in bestimmtes Format umformen
HEX$	Zahl in hexadezimales Äquivalent umformen
OCT$	Zahl in oktales Äquivalent umformen
PRINT USING	Zahl mit bestimmtem Format ausgeben

Konstanten

DATA	Konstantenablage
READ	Konstanten auslesen
RESTORE	Lesezeiger ausrichten

Maschinensprache

CALL	Unterprogramm aufrufen
DEF USR	Unterprogramm definieren
USR	Unterprogramm aufrufen
INP	Port auslesen
OUT	in Port schreiben
MEMORY	Programmeinstellungen und Speichergrenze ändern
OPTION INPUT	Adresse von **Maschinenprogramm INPUT** festlegen
OPTION LPRINT	Adresse von **Maschinenprogramm LPRINT** festlegen
OPTION PRINT	Adresse von **Maschinenprogramm PRINT** festlegen
PEEK	Speicher auslesen
POKE	in Speicher schreiben
VARPTR	Speicherstelle von Variablen ermitteln

Mathematische Funktionen

ABS	Betrag
ATN	Arcus Tangens
COS	Cosinus
EXP	Exponentialfunktion
LOG	natürlicher Logarithmus
LOG10	Logarithmus zur Basis 10
MOD	Modula
SGN	Signum

SIN	Sinus
SQR	Quadratwurzel
TAN	Tangens

Programmierung

AUTO	automatische Zeilennummerierung
CHAIN	Programmverknüpfung
CHAIN MERGE	Programmverknüpfung
MERGE	Programmverknüpfung
DELETE	Programmbereich löschen
EDIT	Zeile ändern
LIST	Programmbereich auflisten
LLIST	Programmbereich ausdrucken
RENUM	Programm neu nummerieren

Programmsteuerung

CONT	Programm nach Unterbrechung fortsetzen
END	Programmende
NEW	Programm komplett löschen
OPTION RUN	Programm kann nicht mit einer Tastenkombination angehalten werden
OPTION STOP	hebt OPTION RUN auf
REM	Programmanmerkungen
RUN	Programm starten
STOP	Programm anhalten
SYSTEM	BASIC beenden und zu CP/M Plus springen

Schleifen

FOR	Schleife mit n Programmschritten
NEXT	Ende der FOR-Schleife
WHILE	Schleife solange Bedingung wahr
WEND	Ende der WHILE-Schleife

Speicherplatz

CLEAR	Variablen löschen
FRE	freien Speicherplatz ermitteln
HIMEM	obere Speichergrenze ermitteln
MEMORY	obere Speichergrenze festlegen

Sprünge

GOSUB	Unterprogramm aufrufen
RETURN	Unterprogramm beenden
GOTO	Zeile anspringen
ON x GOSUB	bestimmtes Unterprogramm aufrufen
ON x GOTO	bestimmte Zeile anspringen

Stringumwandlung in Zahl

CVD	String in doppelt genaue Zahl
CVI	String in Integerzahl
CVIK	Schlüsselstring in Integerahl
CVS	String in einfach genaue Zahl
CVUK	Schlüsselstring in Integerzahl

Stringverarbeitung

ASC	ASCII-Wert eines Zeichens ermitteln
CHR$	Zeichen mit best. ASCII-Code darstellen
INSTR	Zeichenkette in einem String suchen
LEFT$	linken Teilstring ermitteln
LEN	Länge eines Strings ermitteln
LOWER$	String in Kleinbuchstaben umwandeln
MID$	Teilstring ermitteln
RIGHT$	rechten Teilstring ermitteln
SPACE$	Leerstellen erzeugen
STR$	Umwandlung einer Zahl in einen String
STRING$	n Zeichen eines Zeichens darstellen
STRIP$	7. Bit eines Zeichens auf 0 setzen
SWAP	Inhalt zweier Variablen ohne dritte austauschen
UPPER$	String in Großbuchstaben umwandeln
VAL	Wert eines Zahlenstringes ermitteln

Voreinstellungen

COMMON	Variablen als konsistent definieren
COMMON RESET	inkonsistente Variablen löschen
DEF FN	Benutzerfunktion definieren
DEFINT	Integervariablen definieren
DEFSNG	einfach genaue Variablen definieren
DEFDBL	doppelt genaue Variablen definieren
DEFSTR	Stringvariablen definieren
DIM	Felder dimensionieren
ERASE	dimensionierte Felder löschen
OPTION BASE	Feldanfang festlegen
OPTION NOT TAB	Tabulator setzen nicht möglich
OPTION TAB	Tabulator setzen möglich

Zahlenumwandlung in String

MKD$	Zahl in doppelt langen String
MKI$	Zahl in String
MKIK$	Integerzahl in Schlüsselstring
MKS$	Zahl in String umwandeln
MKUK$	Integerzahl in Schlüsselstring

Zahlenverarbeitung

CDBL	Zahl in doppelt genaue Zahl umwandeln
CINT	Zahl in ganze Zahl umwandeln
CSNG	Zahl in einfach genaue Zahl umwandeln
FIX	Zahl in Richtung Null runden
INT	Nachkommastellen einer Zahl abschneiden
MAX	größte Zahl aus mehreren heraussuchen
MIN	kleinste Zahl aus mehreren heraussuchen
ROUND	Zahl runden
SWAP	Inhalt zweier Variablen ohne dritte austauschen
UNT	Zahl in vorzeichenlose Integerzahl umwandeln
VAL	Wert eines Zahlenstringes ermitteln

Zeicheneingabe

INKEY$	einzelnes Zeichen von der Tastatur lesen
INPUT	Zeichenkette von der Tastatur lesen
INPUT$	n Zeichen von der Tastatur lesen
LINE INPUT	komplette Zeichenfolge bis zum nächsten RETURN von der Tastatur lesen

Zufall

RANDOMIZE	Zufallsgenerator initialisieren
RND	Zufallszahl erzeugen

Detaillierte Erläuterungen aller BASIC-Befehle

ABS

ABS (x) gibt den absoluten Wert von **x** zurück. Ein negatives **x** wird durch **ABS (x)** positiv, ein positives **x** positiv übergeben.

Beispiel:

```
10 x=-1.23456
30 PRINT "ABS (";x;") ergibt";ABS (x)
40 PRINT "ABS (4-14) ergibt";ABS (4-14)
50 END
```

RUN:

```
ABS (-1.23456 ) ergibt 1.23456
ABS (4-14) ergibt 10
Ok
■
```

ADDKEY - ADDREC

Bei diesen Kommandos handelt es sich um **JETSAM**-Befehle. Genaueres erfahren Sie im **JETSAM**-Teil auf den Seiten 175 und 177.

ASC

ASC (zeichen$) übergibt den ASCII-Code des ersten Zeichens von **zeichen$**. Der String-Ausdruck **zeichen$** muß mindestens die Länge eines Zeichens haben. Falls man nicht das erste Zeichen von **zeichen$** benötigt, muß man das entsprechende Zeichen mit der Funktion **MID$** (s. dort) heraussuchen.

Beispiel:

```
10 a$="a"                                                    |
20 b$=CHR$(164)                                              | CHR$(164)=
30 c$="Test"                                                 |
40 d$=MID$(c$,2,1)                                           | d$="e" (s. MID$)
50 PRINT "ASC (";CHR$(34);a$;CHR$(34);") ergibt";ASC(a$)     | CHR$(34)="
60 PRINT "ASC (";CHR$(34);b$;CHR$(34);") ergibt";ASC(b$)     |
70 PRINT "ASC (";CHR$(34);c$;CHR$(34);") ergibt";ASC(c$)     |
80 PRINT "ASC (";CHR$(34);d$;CHR$(34);") ergibt";ASC(d$)     |
90 END                                                       |
```

RUN:

```
ASC ("a") ergibt 97                                          |
ASC (" ") ergibt 164                                         |
ASC ("Test") ergibt 84                                       | ASC("T")=84
ASC ("e") ergibt 101                                         | 2. Zeich. von Test
Ok
■
```

Die Gegenfunktion zu **ASC** ist **CHR$**. Mit ihr ist es möglich, das zum entsprechenden ASCII-Code gehörige Zeichen darzustellen (genaue Beschreibung siehe dort).

Im Anhang dieses Buches befindet sich eine vollständige ASCII-Code Tabelle, die auch den ASCII-Bereich 128-159 (Grafik-Sonderzeichen) wiedergibt.

ATN

ATN (x) übergibt den **Arcus Tangens** von **x**. **x** muß dabei größer als $-1.70141178*10^{38}$ und kleiner als $1.70141178*10^{38}$ sein. Zu beachten ist, daß sich alle Angaben auf das **Bogenmaß** beziehen.

Beispiel:

```
10 x=0.546302
20 y=-0.546302
30 PRINT "ATN (";x;") ergibt ";ATN (x)
40 PRINT "ATN (";y;") ergibt ";ATN (y)
50 END
```

RUN:

```
ATN ( 0.546302 ) ergibt  0.5
ATN (-0.546302 ) ergibt -0.5
Ok
■
```

Besonderheiten:

Die Kreisfunktion π (Pi) läßt sich wie folgt berechnen:

```
10 pi=4*ATN(1)
20 PRINT pi
```

A U T O

Durch **AUTO zeilennummer,steprate** wird BASIC veranlaßt, bei der Erstellung eines Programms nach jedem [RETURN] eine neue Zeilennummer zu erzeugen.

Für **zeilennummer** ist dabei die Zeilennummer einzusetzen, bei der die automatische Zeilennummerierung beginnen soll. Wird sie weggelassen, wird als erste Zeilennummer 10 angenommen.

Als **steprate** muß der Abstand von der letzten zur folgenden Zeilennummer angegeben werden. Wird sie weggelassen, wird als Steprate 10 angenommen. Falls das Komma, aber nicht die Steprate vorhanden ist, wird die Steprate des letzten **AUTO**-Kommandos benutzt. Ist es das erste **AUTO**-Kommando, wird auch hier 10 angenommen.
Eine Steprate von **0** ist nicht zulässig.

Nach Erscheinen der neuen Zeilennummer kann wie gewohnt programmiert werden.

Die automatische Zeilennummerierung läßt sich mit [STOP] oder mit [ALT]+[C] abbrechen.

Beispiel:

```
AUTO

AUTO 0,5

AUTO 30,

AUTO 100
```

B U F F E R S

Bei diesem Kommando handelt es sich um einen **JETSAM**-Befehl. Genaueres erfahren Sie im **JETSAM**-Teil auf Seite 168.

C A L L

Durch **CALL adr(HL,DE,BC)** ist es möglich, eigene Maschinenprogramme von BASIC aus aufzurufen.
adr muß vor dem **CALL**-Aufruf definiert werden. Es ist zweckmäßig, dies in hexadezimaler Schreibweise zu tun, z.B. adr=**&HF400**.
HL,DE,BC übergibt beim Aufruf die entsprechenden Werte den entsprechenden Doppelregistern. **HL,DE,BC** müssen ebenfalls, falls sie benutzt werden, als Variablen übergeben werden.

Beispiele:

```
10 MEMORY &HF3FF:screen=&HF400                          | obere Speichergrenze
                                                        | absenken
20 FOR i=&HF400 TO &HF400+63                            |
30  READ a$                                             | Maschinenprogramm
40  wert=VAL("&H"+a$)                                   | in den
50  control=control+wert                                | Speicher
60  POKE i,wert                                         | übertragen
70 NEXT                                                 |
80 IF control<>5187 THEN PRINT "DATAFEHLER !!!":END     |
90 CALL screen                                          | Programm aufrufen
100 END
110 DATA 0E,09                                          | LD C,9
120 DATA 11,09,F4                                       | LD DE,text
130 DATA CD,05,00                                       | CALL BDOS
140 DATA C9                                             | RET
150 DATA 1B,45,1B                                       | text: 1BH,'E',1BH,
160 DATA 48,1B,59,30                                    | 'H',1BH,'Y',30H,
170 DATA 37,44,69,65,73,20,69,73,74,20,65,69,6E         | 37H,'Dies ist ein
180 DATA 20,6D,69,74,20,43,41,4C,4C,20,61,75,66         | mit CALL auf
190 DATA 67,65,72,75,66,65,6E,65,73,20,50,72,6F         | gerufenes Pro
200 DATA 67,72,61,6D,6D,2E,0D,0A,24                     | gramm.',0DH,0AH,'$'
```

RUN:

```
Dies ist ein mit CALL aufgerufenes Programm.
Ok
■
```

C D B L

Die Funktion **CDBL (x)** gibt für den numerischen Ausdruck **x** eine doppelt genaue Zahl zurück.
In der Praxis hat diese Funktion jedoch keine Bedeutung, da sie nicht hinreichend genau arbeitet.

Beispiel:

```
10 a=1.234                          |
20 b#=CDBL (a)                      | Das # steht für die Kennzeichnung einer
30 PRINT "Aus";a;"wird";b#          | doppeltgenauen Variable
40 END
```

RUN:

```
Aus 1.234 wird 1.233999967575073
Ok
■
```

C H A I N

Mit **CHAIN datei$,zeilennummer,ALL** läßt sich aus einem laufenden Programm ein anderes anbinden, wobei alle Variablen erhalten bleiben können.

Die Variable **datei$** gibt das zu ladende Programm an. Ist in **datei$** keine Extension zugeordnet, wird **.BAS** verwendet.

zeilennummer gibt die Zeile an, bei der nach dem Laden das Programm beginnen soll. Wird **zeilennummer** nicht angegeben, beginnt die Ausführung mit der niedrigsten Zeilennummer, bei **zeilennummer=65535** geht das Programm in das Betriebssystem CP/M+ zurück.

Bei Angabe von **ALL** bleiben alle Variablen des laufenden Programms erhalten. Wird **ALL** nicht angegeben, werden alle Variablen gelöscht, sofern sie nicht mit **COMMON** entsprechend gesichert wurden.

Besonderheiten:

- alle Zeilen des aktuellen Programms werden gelöscht
- alle Einstellungen für **DEFINT**, **DEFSNG**, **DEFDBL** und **DEFSTR** werden gelöscht
- die **OPTION BASE**-Spezifikation wird gelöscht, wenn nicht Felder in das neue Programm übertragen wurden.
- alle Benutzerfunktionen (**DEF FN**) werden vergessen
- **ON ERROR GOTO** wird abgeschaltet
- offene Dateien werden nicht geschlossen
- **RESTORE** wird ausgeführt
- alle aktiven **FOR-**, **WHILE-** und **GOSUB**-Kommandos werden vergessen

Beispiele:

```
CHAIN "TEST"                    | Das Programm TEST.BAS wird zum aktiven
                                | Programm herangebunden und gestartet. Alle
                                | Variablen sind gelöscht.

CHAIN "TEST",,ALL               | Das Programm TEST.BAS wird zum aktiven
                                | Programm herangebunden und gestartet. Alle
                                | Variablen bleiben bestehen.

CHAIN "TEST",10,ALL             | Das Programm TEST.BAS wird zum aktiven
                                | Programm herangebunden und  ab Zeile 10
                                | gestartet. Alle Variablen bleiben beste-
                                | hen.
```

C H A I N M E R G E

Mit **CHAIN MERGE datei$,zeilennummer,ALL,DELETE zeilennummerbereich** läßt sich an ein laufendens Programm ein anderes anbinden.

Die Variable **datei$** gibt das zu ladende Programm an. Ist in **datei$** keine Extension zugeordnet, wird **.BAS** verwendet.

zeilennummer gibt die Zeile an, bei der nach dem Laden das Programm weitergeführt werden soll. Wird **zeilennummer** nicht angegeben, beginnt die Ausführung mit der niedrigsten Zeilennummer, bei **zeilennummer=65535** geht das Programm in das Betriebssystem CP/M+ zurück.

Bei Angabe von **ALL** bleiben alle Variablen des laufenden Programms erhalten. Wird **ALL** nicht angegeben, werden alle Variablen gelöscht, sofern sie nicht mit **COMMON** entsprechend gesichert wurden.

DELETE zeilennummerbereich veranlaßt die Löschung der mit **DELETE** angegebenen Zeilen, bevor das neue Programm geladen wird. **zeilennummerbereich** spezifiziert alle Zeilen, deren Nummern im gegebenen Bereich liegen. Dieser Bereich kann eines der folgenden Formate annehmen:

zeilennummer	nur diese Nummer fällt in den Bereich
zeilennummer-zeilennummer	Zeilen von der ersten Nummer bis einschließlich der letzten
zeilennummer-	Zeilen von der bestimmten Zeile bis zum Ende des Programms
-zeilennummer	Zeilen vom Programmanfang bis zu der bestimmten Zeile

Besonderheiten:

- nur die in **DELETE** genannten Zeilen werden gelöscht
- alle Einstellungen für **DEFINT**, **DEFSNG**,**DEFDBL** und **DEFSTR** **bleiben erhalten**
- **OPTION BASE**-Spezifikation bleibt erhalten
- alle Benutzerfunktionen (**DEF FN**) werden vergessen
- **ON ERROR GOTO** wird abgeschaltet
- offene Dateien werden nicht geschlossen
- **RESTORE** wird ausgeführt
- alle aktiven **FOR**-, **WHILE**- und **GOSUB**-Kommandos werden vergessen

Beispiele:

```
CHAIN MERGE "TEST",100,,DELETE 100-200     | Die Zeilen 100 bis 200 werden gelöscht. Das
                                           | Programm TEST.BAS wird herangebunden und ab
                                           | Zeile 100 gestartet. Die Variablen werden
                                           | gelöscht.

CHAIN MERGE "M:TEST.BLD",,ALL,DELETE 1000- | Das aktive Programm wird ab Zeile 1000
                                           | gelöscht. Das Programm TEST.BLD wird
                                           | herangebunden. Die Variablen bleiben
                                           | erhalten.
```

C H R $

Die Funktion **CHR$ (x)** übergibt das dem ASCII-Code **x** zugehörige Zeichen. Das ganzzahlige **x** muß dabei zwischen 0 und 255 liegen.

Zeichen, deren ASCII-Code kleiner als 32 ist, können mit **CHR$(27)+CHR$(x) dargestellt werden.**

CHR$ (x) verhält sich zur Funktion **ASC (x$)** umgekehrt.
Eine vollständige ASCII-Code-Tabelle (ink. Grafik-Sonderzeichen) befindet sich im Anhang des Buches.

Beispiel:

```
10 a=97
20 b=164
30 c=84
40 PRINT "CHR$ (";a;") ergibt ";CHR$(a)
50 PRINT "CHR$ (";b;") ergibt ";CHR$(b)
60 PRINT "CHR$ (";c;") ergibt ";CHR$(c)
70 END
```


RUN:

```
CHR$ ( 97 ) ergibt a
CHR$ ( 164 ) ergibt
CHR$ ( 84 ) ergibt T
Ok
■
```

C I N T

Die Funktion **CINT (x)** rundet den Wert **x** auf eine Integerzahl. **x** muß dabei zwischen -32768 und +32767 liegen.
Werden diese Grenzen nicht beachtet, bricht BASIC das Programm mit der Fehlermeldung **Improper argument** ab.

Beispiel:

```
10 a=123.456
20 b=-456.789
30 PRINT "CINT (";a;") ergibt ";CINT (a)
40 PRINT "CINT (";b;") ergibt ";CINT (b)
50 END
```

RUN:

```
CINT ( 123.456 ) ergibt  123
CINT (-456.789 ) ergibt -457
OK
■
```

C L E A R

CLEAR,High-Memory,Stack-Größe, Dateizahl, maximale Satzlänge löscht alle Variablen, vergißt alle aktuellen **FOR**-, **WHILE**- und **GOSUB**-Kommandos, schließt alle offenen Dateien und verändert die angegebenen Parameter.

High-Memory bezieht sich auf die obere vom BASIC benutzbare Speichergrenze; **High-Memory** muß somit in der Form einer Adresse eingegeben werden. Wird diese Angabe weggelassen, bleibt der aktuelle Wert eingestellt.

Stack-Größe gibt die Anzahl der von BASIC frei für das Stapelregister verwendbaren Bytes an. Der Stack ist der Speicherbereich, der von BASIC für die Verwaltung von Schleifen, **GOSUB**-Routinen, etc. benötigt wird. Falls **Stack-Größe** angegeben wird, muß mindestens ein Wert von 256 verwendet wer-

den. Wird diese Angabe weggelassen, bleibt der aktuelle Wert eingestellt.

Dateizahl gibt die maximale Anzahl gleichzeitig zu bearbeitender Dateien an. Ohne diese Angabe bleibt der aktuelle Wert eingestellt.

maximale_Satzlänge gibt die maximale Größe eines frei adressierbaren Satzes an. Ohne diese Angabe bleibt der aktuelle Wert eingestellt.

Die Standardeinstellungen:

High-Memory	&HF605
Stack-Größe	512 Bytes
Dateizahl	3
max._Satzlänge	128 Bytes

Beispiel:

```
CLEAR                    | nur Variablen werden gelöscht

CLEAR,&HF605,,6          | alle Variablen werden gelöscht, High-Memory auf
                         | &HF605, 6 Dateien

CLEAR,&HEFFF,512,3,256   | alle Variablen werden gelöscht, High-Memory auf
                         | &HEFFF, Stack-Größe auf 512 Bytes, 3 Dateien,
                         | 256 Bytes maximale Satzlänge

CLEAR,,,10               | alle Variablen werden gelöscht, 10 Dateien

10 a=100                 | Initialisierung
20 b=200                 |
30 PRINT "Vor CLEAR:"    |
40 PRINT a;b             | Kontrollabfrage: a,b erhalten
50 CLEAR                 | alle Variablen löschen
60 PRINT "Nach CLEAR:"   |
70 PRINT a;b             | Kontrollabfrage: a,b gelöscht
80 END                   |
```

RUN:

```
Vor CLEAR:
 100  200
Nach CLEAR:
 0  0
```

Hinweis: Ein weiteres Kommando zum Ändern der Parameter ist **MEMORY**. Hierbei werden jedoch keine Variablen gelöscht.

C L O S E

Durch **CLOSE dateinummerliste** werden die mit **dateinummerliste** spezifizierten Dateien geschlossen.

dateinummerliste ist in der Form **#nr_a,#nr_b,#nr_c,..** anzugeben. Wird **dateinummerliste** weggelassen, werden sämtliche offenen Dateien geschlossen.

Beispiel:

```
10 OPEN "O",#1,"M:TEST1"                          | Datei öffnen: #1
20 OPEN "O",#2,"M:TEST2"                          | Datei öffnen: #2
30 PRINT #1,"Dies ist eine Testzeile für Datei 1." | Test für #1
40 PRINT #2,"Dies ist eine Testzeile für Datei 2." | Test für #2
50 CLOSE #1,#2                                    | #1, #2 schließen
60 TYPE M:TEST1                                   | Inhaltskontrolle #1
70 TYPE M:TEST2                                   | Inhaltskontrolle #2
80 END                                            | Ende
```

RUN:

```
Dies ist eine Testzeile für Datei 1.
Dies ist eine Testzeile für Datei 2.
Ok
■
```

C O M M O N

Mit **COMMON variablenliste** können Variablen spezifiziert werden, die während der Ausführung von **COMMON RESET**, **CHAIN** oder **CHAIN MERGE** nicht gelöscht werden sollen. **variablenliste** enthält dabei Variablen jeglichen Typs.
Die Ausführung eines **COMMON**-Befehles im Direktmodus (Zeitpunkt, zu dem man ein Programm eingeben kann; also die Zeit, zu der das Programm nicht abläuft) hat keine Wirkung.

Beispiel:

```
10 COMMON a,a$                          | a,a$ als konsistent definieren
20 a=100                                | Wertzuweisung
30 a$="Test_1"                          | Wertzuweisung
40 b=200                                | Wertzuweisung
50 b$="Test_2"                          | Wertzuweisung
60 PRINT "Vor COMMON RESET: ";a;a$;b;b$ | Kontrolle: a,a$,b,b$          Ok
70 COMMON RESET                         | Inkonsistente Variablen löschen
80 PRINT "Nach COMMON RESET:"a;a$;b;b$  | Kontrolle: a,a$ Ok; b,b$      gelöscht
90 END
```

RUN:

```
Vor COMMON RESET:  100 Test_1 200 Test_2
Nach COMMON RESET: 100 Test_1 0
Ok                 |    |   |   |
■                  a----a$   b---b$
                     |         |
                gesichert   ungesichert
```

C O M M O N R E S E T

Durch Verwendung von **COMMON RESET** werden alle Variablen gelöscht, die nicht durch **COMMON** spezifiziert wurden. Außerdem werden alle aktiven **FOR-**, **WHILE-** und **GOSUB-**Kommandos vergessen.

Beispiel:

```
10 COMMON a,a$                          | a,a$ als konsistent definieren
20 a=100                                | Wertzuweisung
30 a$="Test_1"                          | Wertzuweisung
40 b=200                                | Wertzuweisung
50 b$="Test_2"                          | Wertzuweisung
60 PRINT "Vor COMMON RESET: ";a;a$;b;b$ | Kontrolle: a,a$,b,b$            Ok
70 COMMON RESET                         | Inkonsistente Variablen löschen
80 PRINT "Nach COMMON RESET:"a;a$;b;b$  | Kontrolle: a,a$ Ok; b,b$        gelöscht
90 END
```

RUN:

```
Vor COMMON RESET:  100 Test_1 200 Test_2
Nach COMMON RESET: 100 Test_1 0
Ok                 |    |   |   |
■                  a----a$   b---b$
                     |         |
                gesichert   ungesichert
```

C O N S O L I D A T E

Bei diesem Kommando handelt es sich um einen **JETSAM**-Befehl. Genaueres erfahren Sie im **JETSAM**-Teil auf Seite 172.

C O N T

Wenn das Programm durch irgendeinen Umstand (Break (durch [ALT]+[C] oder [STOP]), **STOP**, **END**, Error) angehalten wurde, kann es durch Eingabe von **CONT** im Direktmodus fortgesetzt werden. Falls das Programm in dieser Pause geändert wurde, wird das **CONT**-Kommando mit der Fehlermeldung **Cannot CONTinue** zurückgewiesen.

Beispiel:

```
10 PRINT "Dies ist ein Programm."
20 STOP
30 PRINT "Dies ist das durch CONT fortgesetzte Programm."
40 END
```

RUN:

```
Dies ist ein Programm.
Break in 20
Ok
■
CONT (muß von Ihnen eingegeben werden)
Dies ist das durch CONT fortgesetzte Programm.
Ok
■
```

C O S

COS (x) übergibt den Cosinus von **x**. **x** muß dabei größer als -205884.2734375 und kleiner als 205884.2734375 sein.
Zu beachten ist, daß sich alle Angaben auf das **Bogenmaß** beziehen.

Beispiel:

```
10 x=3.1415927
20 y=6.2831853
30 PRINT "COS (";x;") ergibt ";COS (x)
40 PRINT "COS (";y;") ergibt ";COS (y)
50 END
```

RUN:

```
COS ( 3.1415927 ) ergibt -1
COS ( 6.2831853 ) ergibt  1
Ok
■
```

C R E A T E

Bei diesem Kommando handelt es sich um einen **JETSAM**-Befehl. Genaueres erfahren Sie im **JETSAM**-Teil auf Seite 170.

C S N G

CSNG (x) übergibt den einfach genauen Wert von **x**.

Beispiel:

```
10 x#=0.12345678901234#                     | #=Kennzeichnung doppelt genaue Zahl
20 PRINT "CSNG (";x#;") ergibt";CSNG (x#)   |
30 END
```

RUN:

```
CSNG ( 0.12345678901234 ) ergibt 0.1234568
Ok
■
```

C V D

CVD (zahl$) wandelt die durch **MKD$** erzeugte Zeichenkette **zahl$** in eine doppelt genaue Zahl zurück.

Beispiel:

```
10 a#=0.1234567890123456#              | #=Kennzeichnung doppelt genaue Zahl
20 zahl$=MKD$ (a#)                     |
30 PRINT "CVD (";CHR$(34);zahl$;CHR$(34);") ergibt";CVD (zahl$)
40 END
```


RUN:

```
CVD ("E¯7¯ЖöU") ergibt 0.1234567890123456
Ok         |                             |
■          |                             |
           |                             |
Dies ist der durch MKD$ (s. dort)| erzeugte 8 Byte lange String. Die hier angezeigten
Zeichen entsprechen den ASCII-Ze | ichen der einzelnen Bytes.
                                 |
      Dies ist die durch CVD aus dem String zurückgewandelte doppelt genaue Zahl.
```

C V I

CVI (integerzahl$) wandelt die durch **MKI$** erzeugte Zeichenkette **integerzahl$** in eine Integerzahl zurück.

Beispiel:

```
10 a!=-1                    | !=Kennzeichnung einer den Integerbereich überschreitenden
20 b!=32767!                |   Zahl
30 zahl1$=MKI$ (a!)
40 zahl2$=MKI$ (b!)
50 PRINT "CVI (";CHR$(34);zahl1$;CHR$(34);") ergibt ";CVI (zahl1$)
60 PRINT "CVI (";CHR$(34);zahl2$;CHR$(34);") ergibt ";CVI (zahl2$)
70 END
```

RUN:

```
CVI ("≢≡") ergibt -1
CVI ("■0") ergibt  32767
Ok    |                 |
■     |                 |
      |                 |
Dies ist der durch M | KI$ (s. dort) erzeugte 2 Byte lange String. Die hier
angezeigten Zeichen  | entsprechen den ASCII-Zeichen der einzelnen Bytes.
                     |
      Dies ist die durch CVI aus dem String zurückgewandelte Integerzahl.
```

C V I K

CVIK (integerzahl$) wandelt die durch **MKIK$** erzeugte Zeichenkette **integerzahl$** in eine Integerzahl zurück.

Beispiel:

```
10 a!=-1                    | !=Kennzeichnung einer den Integerbereich überschreitenden
20 b!=32767!                |   Zahl
30 zahl1$=MKIK$ (a!)
40 zahl2$=MKIK$ (b!)
50 PRINT "CVIK (";CHR$(34);zahl1$;CHR$(34);") ergibt ";CVIK (zahl1$)
60 PRINT "CVIK (";CHR$(34);zahl2$;CHR$(34);") ergibt ";CVIK (zahl2$)
70 END
```

RUN:

```
CVIK ("0=") ergibt -1
CVIK ("==") ergibt  32767
Ok      |           |
■       |           |
        |           |
Dies ist der durch M | KIK$ (s. dort) erzeugte 2 Byte lange String. Die hier
angezeigten Zeichen  | entsprechen den ASCII-Zeichen der einzelnen Bytes.
                     |
        Dies ist die durch CVIK aus dem String zurückgewandelte Integerzahl.
```

C V S

CVS (zahl$) wandelt die durch **MKS$** erzeugte Zeichenkette **zahl$** in eine einfach genaue Zahl zurück.

Beispiel:

```
10 a=12345.67
20 b=-12345.67
30 zahl1$=MKS$ (a)
40 zahl2$=MKS$ (b)
50 PRINT "CVS (";CHR$(34);zahl1$;CHR$(34);") ergibt ";CVS (zahl1$)
60 PRINT "CVS (";CHR$(34);zahl2$;CHR$(34);") ergibt ";CVS (zahl2$)
70 END
```


RUN:

```
CVS ("¿ê$=") ergibt  12345.67
CVS ("¿êA=") ergibt -12345.67
Ok      |           |
■       |           |
        |           |
Dies ist der durch MKS$ | (s. dort) erzeugte 4 Byte lange String. Die hier ange-
zeigten Zeichen entspre | chen den ASCII-Zeichen der einzelnen Bytes.
                        |
        Dies ist die durch CVS aus dem String zurückgewandelte einfach genaue Zahl.
```

C V U K

CVUK (integerzahl$) wandelt die durch **MKUK$** erzeugte Zeichenkette **integerzahl$** in eine Integerzahl zurück.

Beispiel:

```
10 a=12345
20 b=54321!
30 zahl1$=MKUK$ (a)
40 zahl2$=MKUK$ (b)
50 PRINT "CVUK (";CHR$(34);zahl1$;CHR$(34);") ergibt";CVUK (zahl1$)
60 PRINT "CVUK (";CHR$(34);zahl2$;CHR$(34);") ergibt";CVUK (zahl2$)
70 END
```

RUN:

```
CVUK ("09") ergibt 12345
CVUK ("11") ergibt 54321
Ok      |           |
■       |           |
        |           |
Dies ist der durch M | KUK$ (s. dort) erzeugte 2 Byte lange String. Die hier
angezeigten Zeichen  | entsprechen den ASCII-Zeichen der einzelnen Bytes.
                     |
        Dies ist die durch CVUK aus dem String zurückgewandelte Integerzahl.
```

D A T A

Dieser Befehl wird in der Form **DATA konstantenliste** verwendet. **konstantenliste**, die eine programminterne Ablage von Konstanten darstellt, kann sich dabei aus Zahlen oder Buchstaben zusammensetzen. Alle aufeinanderfolgenden Elemente müssen dabei durch Komma getrennt werden. Leerzeichen vor oder nach einem Element werden ignoriert.
Falls in einem String Kommas oder Doppelpunkte vorkommen, muß der String mit Anführungszeichen eingegeben werden, da BASIC sonst das Komma bzw. den Doppelpunkt als Ende des Strings interpretiert. Das abschließende Anführungszeichen kann weggelassen werden, wenn es das letzte Zeichen einer Zeile ist.

Beispiel:

```
10 RESTORE 130                          | DATA-Zeiger auf Zeile 130 richten
20 FOR i=1 to 3
30  READ a
40  PRINT a
50 NEXT
60 PRINT
70 RESTORE 140                          | DATA-Zeiger auf Zeile 140 richten
80 FOR i=1 to 4
90  READ a$
100  PRINT a$
110 NEXT
120 END
130 DATA 10,-1.234,-97531
140 DATA Test,Autobahnraststätte
150 DATA "Kommatest,Doppelpunkttest:","6,19,23,30,36,45 / 5
```

RUN:

```
 10
-1.234
-97531

Test
Autobahnraststätte
Kommatest,Doppelpunkttest:
6,19,23,30,36,45 / 5
Ok
■
```


D E C $

DEC$(a,format$) wird für die Umwandlung der Zahl **a** in das durch **format$** angegebene Format verwendet. **a** ist dabei ein beliebiger numerischer Ausdruck.

Für **format$** sind verschiedene Optionen einsetzbar:
Jedes **#** repräsentiert die Position einer Ziffer. In **format$** muß mindestens ein **#** enthalten sein.
. legt die Position des Dezimalpunktes fest. In **format$** darf höchstens ein **.** enthalten sein.
Ein **,** bewirkt die Aufteilung der Vorkommastellen in Dreiergruppen, die durch Komma getrennt werden. Das **,** muß vor den **.** gesetzt werden.
Mit **$$** kann man vor die erste Ziffer bzw. den Dezimalpunkt ein **$** setzen. Die **$$** müssen vor dem Zahlenformat stehen.
Bei Verwendung von ****** werden führende Leerstellen mit Sternen aufgefüllt. Die ****** müssen vor dem Zahlenformat stehen.
Durch ****$** werden **$$** und ****** kombiniert. Auch ****$** muß vor dem Zahlenformat stehen.
Mit **+** wird das Vorzeichen der Zahl ausgegeben. Steht **+** vor dem Zahlenformat, wird das Vorzeichen vor die Zahl oder das **$** gestellt. Steht **+** am Ende des Zahlenformats, wird das Vorzeichen nachgestellt.
Ein **-** bewirkt die nachgestellte Ausgabe eines Minuszeichens oder eines Leerzeichens für positive Zahlen. **-** muß nach dem Zahlenformats stehen.
Bei Verwendung von **↑↑↑↑** wird die Zahl in Exponentialdarstellung ausgegeben. **↑↑↑↑** muß direkt hinter dem Zahlenformat stehen.

Beispiel:

```
10 DEFDBL a                             | a=doppelt genau
20 a=-123456.789#                       | # wird von BASIC gesetzt
30 PRINT DEC$(a,"########")             | im folgenden werden die
40 PRINT DEC$(a,"########.######")      | verschiedensten Formate durch-
50 PRINT DEC$(a,"########,.######")     | getestet
60 PRINT DEC$(a,"$$########,.######")   |
70 PRINT DEC$(a,"$$########.######")    |
80 PRINT DEC$(a,"***########,.######")  |
90 PRINT DEC$(a,"***########.######")   |
100 PRINT DEC$(a,"***$########,.######") |
110 PRINT DEC$(a,"***$########.######")  |
120 PRINT DEC$(a,"+$$########,.######")  |
130 PRINT DEC$(a,"$$########.######+")   |
140 PRINT DEC$(a,"$$########.######-")   |
150 PRINT DEC$(a,"$$########.######↑↑↑↑-") |
160 END
```

RUN:

```
 -123457
 -123456.789000
 -123,456.789000
  -$123,456.789000
  -$123456.789000
***-123,456.789000
***-123456.789000
***-$123,456.789000
***-$123456.789000
   -$123456.789000
   $123456.789000-
   $123456.789000-
$123456789.0000000D-03-
Ok
■
```

D E F F N

Mit **DEF FNx(a,b,c,...)=funktion(a,b,c,...)** (**DEF**ine **FN**ktion) läßt sich eine Funktion definieren, die bei späterer Gelegenheit universell aufgerufen werden kann. Prinzipiell gibt es vier verschiedene Grundmöglichkeiten, das **DEF FN**-Kommando einzusetzen:

Einfache numerische Verwendung
`DEF FNx=a*b-c+d`: Nach Aufruf durch **wert=FNx** wird **a*b-c+d** ausgerechnet und **wert** übergeben.

Numerische Verwendung mit Variablenübergabe
`DEF FNx(y)=(y+1)↑2`: Nach Aufruf durch **wert=FNx(y)** wird die Funktion **(y+1)↑2** ausgerechnet und das Ergebnis **wert** Übergeben.

Einfache Stringverwendung
`DEF FNx$=a$+b$`: Nach Aufruf durch **wert$=FNx$** wird ein String, der aus **a$+b$** besteht, an **wert$** übergeben.

Stringverwendung mit Variablenübergabe
`DEF FNx$(y)=a$(y)+b$(y)`: Nach Aufruf durch **wert$=FNx$(y)** wird ein String, der aus **a$(y)+b$(y)** besteht, **wert$** übergeben.

Der Typ der Funktion muß mit dem Typ der Variable kompatibel sein (numerisch oder String). Im Übrigen kann die Funktion auch andere Funktionen aufrufen. Im Direktmodus ist das Kommando ungültig.

Beispiel:

```
10 DEF FNa(x)=x^2-4*x+2
20 DEF FNb(x)=FNa(x)+10
30 DEF FNy$(x)=STRING$(x,"J")
40 DEF FNz$(x,y)=STRING$(x,CHR$(y))
50 DEF FNlocate$(x,y)=CHR$(27)+"Y"+CHR$(32+x)+CHR$(32+y)
60 PRINT CHR$(27)+"E"+CHR$(27)+"H"
70 PRINT "FNa(2)=";FNa(2)
80 PRINT "FNb(2)=";FNb(2)
90 PRINT "FNy$(10)=";FNy$(10)
100 PRINT "FNz$(10,255)=";FNz$(10,255)
110 PRINT "FNlocate$(0,0)+FNz$(10,255)=";FNlocate$(0,0)+FNz$(10,255)
120 END
```

| verschiedene Funktionen definieren

| (Cursor positionieren) Bildschirm löschen Funktionen mit Werten testen

RUN:

```
(Bildschirm gelöscht)
FNa(2)=-2
FNb(2)= 8
FNy$(10)=JJJJJJJJJJ
FNz$(10,255)=
FNlocate$(0,0)+FNz$(10,255)=


Ok
■
```

D E F U S R

DEF USR nummer=adresse legt bis zu 10 (0-9) verschiedene Maschinen-Unterprogramme fest. **nummer** bezeichnet dabei das Unterprogramm, **adresse** den entsprechenden Beginn desselben. Wird **nummer** nicht angegeben, wird der Wert 0 angenommen. Das Unterprogramm wird durch die Funktion **USR** aufgerufen.

Beispiele:

```
10 MEMORY &HF3FF:DEF USR 1=&HF400
20 FOR i=&HF400 TO &HF400+62
30  READ a$
40  wert=VAL("&H"+a$)
50  POKE i,wert
60 NEXT
70 a=USR 1(b) 'a und b stellen Hilfsvariablen dar, um das Maschinenunterprogramm
              'über die USR-Funktion zu starten.
80 END
90 DATA 0E,09                                   | LD C,9
100 DATA 11,09,F4                               | LD DE,text
110 DATA CD,05,00                               | CALL BDOS
```

```
120 DATA C9                                          | RET
130 DATA 1B,45,1B,48                                 | text: 1BH,'E',1BH,'H',
140 DATA 1B,59,30,37,44,69,65,73,20                  | 1BH,'Y',30H,37H,'Dies
150 DATA 69,73,74,20,65,69,6E,20,6D,69,74,20,55,53,52 | ist ein mit USR
160 DATA 20,61,75,66,67,65,72,75,66,65,6E,65,73,20,50 |  aufgerufenes P
170 DATA 72,6F,67,72,61,6D,6D,2E,0D,0A,24            | rogramm.',0DH,0AH,'$'
```

RUN:

```
Dies ist ein mit USR aufgerufenes Programm.
Ok
■
```

D E F D B L

Mit **DEFDBL buchstabenbereich** lassen sich die mit **buchstabenbereich** bezeichneten Variablen als doppelt genaue Variablen vordefinieren.
buchstabenbereich muß entweder in der Form eines einzelnen Buchstabens eingegeben werden oder in der Form einer Buchstabenliste, also Buchstabe1-Buchstabe2 (einschließlich).
In der Praxis hat dieses Kommando jedoch keine Bedeutung, da Mallard-80 BASIC mit doppelt genauen Zahlen nicht hinreichend genau arbeitet.

Beispiel:

```
DEFDBL b                | b ist doppelt genau

DEFDBL e-z              | die Variablen e bis z sind doppelt genau

10 DEFDBL b
20 a=1.234
30 b=CDBL(a)
40 PRINT "Aus";a;"wird";b
50 END
```

RUN:

```
Aus 1.234 wird 1.233999967575073
Ok
■
```


D E F I N T

Mit **DEFINT buchstabenbereich** lassen sich die mit **buchstabenbereich** bezeichneten Variablen als Integervariablen vordefinieren.
buchstabenbereich muß entweder in der Form eines einzelnen Buchstabens eingegeben werden oder in der Form einer Buchstabenliste, also Buchstabe1-Buchstabe2 (einschließlich).
Eine Integervariable ist eine Variable, die nur ganzzahlige Werte im Bereich von -32768 bis +32767 annehmen kann.
Durch eine Voreinstellung mit **DEFINT** läßt sich die Rechenzeit eines Programmes erheblich verkürzen, was sich besonders bei **FOR..NEXT**-Schleifen positiv bemerkbar macht.

Beispiel:

```
DEFINT b                | b ist als integer definiert

DEFINT e-z              | die Variablen e bis z sind als integer definiert

10 DEFINT b-c
20 a=1.23456
30 b=1.23456
40 c=4.99999
50 PRINT "a (einfach genau) hat den Wert";a
60 PRINT "b (integer) hat den Wert";b
70 PRINT "c (integer) hat den Wert";c
80 END
```

RUN:

```
a (einfach genau) hat den Wert 1.23456
b (integer) hat den Wert 1
c (integer) hat den Wert 5
Ok
■
```

D E F S N G

Mit **DEFSNG buchstabenbereich** lassen sich die mit **buchstabenbereich** bezeichneten Variablen als einfach genaue Variablen vordefinieren.
buchstabenbereich muß entweder in der Form eines einzelnen Buchstabens oder in der Form einer Buchstabenliste, also Buchstabe1-Buchstabe2 (einschließlich), eingegeben werden.
Die Voreinstellung für alle Variablen ist die einfach genaue Variable. Dieser Zustand wird durch **LOAD**, **RUN**, **CHAIN**, **NEW** und **CLEAR** eingestellt.

Beispiel:

```
DEFSNG b                | b ist als einfach genau definiert

DEFSNG e-z              | die Variablen e bis z sind als einfach genau definiert

10 DEFSNG a-d
20 a=1.23456
30 b=-3.89673
40 c=4.99999
50 d=a*b/c
60 PRINT "a (einfach genau) hat den Wert ";a
70 PRINT "b (einfach genau) hat den Wert ";b
80 PRINT "c (einfach genau) hat den Wert ";c
90 PRINT "d (einfach genau) hat den Wert ";d
100 END
```

RUN:

```
a (einfach genau) hat den Wert  1.23456
b (einfach genau) hat den Wert -3.89673
c (einfach genau) hat den Wert  4.99999
d (einfach genau) hat den Wert -1.5840943
Ok
■
```

D E F S T R

Mit **DEFSTR buchstabenbereich** lassen sich die mit **buchstabenbereich** bezeichneten Variablen als Stringvariablen vordefinieren.
buchstabenbereich muß entweder in der Form eines einzelnen Buchstabens oder in der Form einer Buchstabenliste, also Buchstabe1-Buchstabe2 (einschließlich), eingegeben werden.
Die Folge ist also, daß man Stringvariablen nicht mehr durch ein **$** kennzeichnen muß.

Beispiel:

```
DEFSTR b                | b ist als Stringvariable definiert

DEFSTR e-z              | die Variablen e bis z sind als Stringvariablen definiert

10 DEFSTR b
20 a=1.2345630
30 b="DEFSTR-Test"
40 PRINT "a (einfach genau) hat den Wert";a
50 PRINT "b (String) ist ";b
60 END
```


RUN

```
a (einfach genau) hat den Wert 1.23456
b (String) ist DEFSTR-Test
Ok
■
```

D E L E T E

Durch das Kommando **DELETE zeilennummernbereich, startzeilennummer** lassen sich die mit **zeilennummernbereich** festgelegten Zeilen aus dem aktuellen Programm löschen. Das Programm fährt dann mit der durch **startzeilennummer** bezeichneten Zeile fort.

zeilennummernbereich spezifiziert alle Zeilen, deren Nummern im gegebenen Bereich liegen. Dieser Bereich kann eines der folgenden Formate annehmen:

zeilennummer	nur diese Nummer fällt in den Bereich.
zeilennummer-zeilennummer	Zeilen von der ersten Nummer bis einschließlich der letzten.
zeilennummer-	Zeilen von der bestimmten Zeile bis zum Ende des Programms.
-zeilennummer	Zeilen vom Programmanfang bis zu der bestimmten Zeile.

Falls der Parameter **startzeilennummer** weggelassen wird, kehrt BASIC nach der Ausführung von **DELETE** in den Direktmodus zurück. Wird als **startzeilennummer** 0 angegeben, beginnt BASIC die Ausführung des Programms mit der niedrigsten Zeilennummer.

Als Nebenwirkungen des **DELETE**-Kommandos werden alle Benutzerfunktionen (s. **DEF FN**) vergessen, die **ON ERROR GOTO**-Fehlerbehandlungsroutine wird abgeschaltet, ein **RESTORE** wird durchgeführt und es werden alle aktiven **FOR-**, **WHILE-** und **GOSUB**-Schleifen vergessen.

Beispiel:

```
DELETE 40           | Löschen der Zeile 40

DELETE 10-30        | Löschen der Zeilen 10 bis 30

DELETE 100-         | Löschen des Programms ab Zeile 100

DELETE -100         | Löschen des Programms bis Zeile 100

DELETE 40,50        | Löschen von Zeile 40, fortfahren des Programms mit Zeile 50

DELETE 10-30,40     | Löschen der Zeilen 10 bis 30, fortfahren ab Zeile 40
```

```
DELETE 100-,10        | Löschen des Programms ab Zeile 100, weiter ab Zeile 10

DELETE -100,150       | Löschen des Programms bis Zeile 100, weiter ab Zeile 150
```

D E L K E Y

Bei diesem Kommando handelt es sich um einen **JETSAM**-Befehl. Genaueres erfahren Sie im **JETSAM**-Teil auf Seite 192.

D I M

Mit **DIM variable(a,b,..),..** wird der nötige Speicherbereich für **variable** festgelegt - **variable** wird **DIM**ensioniert. **a,b,..** stellt dabei eine Liste von Integerzahlen dar, die den Maximalwert der entsprechenden Dimension festlegen.
Ohne **DIM**ensionierung beträgt die Voreinstellung für jede Dimension maximal 10. Um sie zu vergrößern oder zu verkleinern (um Speicherplatz einzusparen) muß das **DIM**-Kommando benutzt werden.
Es ist nicht möglich, eine bereits ge**DIM**te Variable erneut zu dimensionieren. Wird es trotzdem versucht, reagiert der Computer mit der Fehlermeldung **Array already dimensioned** (Fehler 10). Wenn der Computer **Subscript out of range** (Fehler 9) meldet, ist der vordimensionierte Bereich unter- (s. **OPTON BASE**) oder überschritten worden.

Beispiel:

```
10 OPTION BASE 1                        | Dimensionen beginnen bei 1
20 DIM a(20,20),b(40)                   | Festlegung der Diemsionen von a und b
30 FOR i=1 to 20                        | Schleifen,
40   FOR u=1 to 20                      | um die Variablen
50       a(i,u)=u                       | mit Werten aufzufüllen
60   NEXT                               | Ende Schleife 2
70 NEXT                                 | Ende Schleife 1
80 FOR i=1 to 40                        | Anfang Schleife 3
90   b(i)=40-i                          | Wertzuweisung
100 NEXT                                | Ende Schleife 3
110 PRINT "a(10,15)=";a(10,15)          | Kontrolle einzelner
120 PRINT "b(19)=";b(19)                | Werte
130 END                                 | Programmende
```


RUN:

```
a(10,15)= 15
b(19)= 21
Ok
■
```

D I R

DIR dateinamenliste listet die mit **dateinamenliste** spezifizierten Dateien auf dem Bildschirm auf, sofern sie sich im Directory des entsprechenden Laufwerkes befinden. **dateinamenliste** ist in der Form **Datei1.Ext,Datei2.Ext,Datei3.Ext,..** einzugeben, wobei auch Wildcards (?,*) auf übliche Weise verwendet werden dürfen.

DIR interpretiert den Rest der aktuellen Zeile als Argument, unabhängig, ob es sich dabei um Dateinamen handelt oder nicht.

Beispiel:

```
DIR BASIC.COM

DIR BASIC.COM,RPED.BAS

DIR *.COM

DIR B???????.C?M
```

Wenn sich **BASIC.COM** auf der Diskette befindet, sollte in allen Fällen mindestens

```
BASIC.COM
Ok
■
```

auf dem Bildschirm erscheinen.

D I S P L A Y

DISPLAY datei$** zeigt den Inhalt von **datei$ an der Konsole. **DISPLAY** arbeitet prinzipiell wie das CP/M-Kommando **TYPE**. Die Auflistung kann duch [ALT]+[C] (STOP) abgebrochen, durch [ALT]+[S] (f3/f4) angehalten und durch [ALT]+[Q] (f5/f6) fortgesetzt werden. Zu beachten ist, daß **datei$** keine Wildcards (?,*) enthalten darf, sondern eine exakte Dateispezifikation darstellen muß.

Beispiel:

```
10 datei$="PROFILE.GER"
20 DISPLAY datei$
30 PRINT
40 DISPLAY "PROFILE.GER"
50 END
```

RUN:
(**PROFILE.GER** muß sich auf der Diskette im aktuellen Laufwerk befinden!)

```
setdef m:,* [order = (sub,com) temporary = m:]
pip
<m:=basic.com[o]
<m:=dir.com[o]
<m:=erase.com[o]
<m:=paper.com[o]
<m:=pip.com[o]
<m:=rename.com[o]
<m:=setkeys.com[o]
<m:=show.com[o]
<m:=submit.com[o]
<m:=type.com[o]
<

setdef m:,* [order = (sub,com) temporary = m:]
pip
<m:=basic.com[o]
<m:=dir.com[o]
     .....
     .....
<m:=type.com[o]
<
Ok
■
```


E D I T

Mit dem Kommando **EDIT zeilennummer** läßt sich die mit **zeilennummer** spezifizierte Programmzeile ändern.

Beispiel:

```
10 a$="TESTT"
20 PRINT a$
30 END
EDIT 10 (muß von Ihnen eingegeben werden)
10 a$="TESTT" (jetzt können Sie die Zeile korrigieren)
```

RUN:

```
TEST (bzw. Ihre Änderung)
Ok
■
```

E N D

Durch das **END**-Kommando wird das Programm an dieser Stelle ordnungsgemäß beendet. Alle noch offenen Dateien werden geschlossen, danach kehrt das Programm in den Direktmodus zurück. Steht am Ende des Programms kein **END**, werden evtl. offene Dateien erst beim nächsten **RUN** oder **SYSTEM** geschlossen.

Es ist ratsam, aus strukturellen Gründen ein Programm grundsätzlich mit **END** abzuschließen, auch wenn es in den meisten Fällen aus programmtechnischen Gründen nicht unbedingt erforderlich ist.

Beispiel:

```
10 'Hier steht Ihr Programm          | Ihr Programm kann in den Zeilen 0-9998
9999 END                             | stehen
```

RUN:

```
Ihr Programm...
Ok
■
```

E O F

EOF (dateinummer) prüft, ob das Dateiende der durch **dateinummer** spezifizierten Datei erreicht wurde. Ist dies der Fall, wird -1, ansonsten 0 übergeben.

Beispiel:

```
10 DIM inhalt$(50)
20 satz=0
30 OPEN "I",#1,"PROFILE.GER"
40 WHILE NOT EOF(1)                    | Grundsatzbeispiel zum Einlesen von
50   satz=satz+1                       | Daten aus einer Datei unbekannter
60   LINE INPUT #1,inhalt$(satz)       | Länge / Dimension / Dateneinträge
70 WEND
80 CLOSE
90 FOR i=1 to satz
100   PRINT "Satz ";DEC$(i,"##");" = ";inhalt$(i)
110 NEXT
120 END
```

RUN:

```
Satz  1 = setdef m:,* [order = (sub,com) temporary = m:]
Satz  2 = pip
Satz  3 = <m:=basic.com[o]
Satz  4 = <m:=dir.com[o]
Satz  5 = <m:=erase.com[o]
Satz  6 = <m:=paper.com[o]
Satz  7 = <m:=pip.com[o]
Satz  8 = <m:=rename.com[o]
Satz  9 = <m:=setkeys.com[o]
Satz 10 = <m:=show.com[o]
Satz 11 = <m:=submit.com[o]
Satz 12 = <m:=type.com[o]
Satz 13 = <
Ok
m
```

E R A

ERA dateinamenliste löscht die mit **dateinamenliste** spezifizierten Dateien. **dateinamenliste** ist in der Form **Datei1.Ext, Datei2.Ext,Datei3.Ext,...** einzugeben, wobei auch Wildcards (?,*) auf übliche Weise verwendet werden dürfen. Hierbei ist jedoch Vorsicht geboten, da keine Sicherheitsabfrage wie unter CP/M Plus stattfindet.

ERA interpretiert den Rest der aktuellen Zeile als Argument, unabhängig ob es sich dabei um Dateinamen handelt oder nicht.

Beispiel: **(ACHTUNG: ES IST SINNVOLL, DIE BEISPIELE NICHT AUSZUPROBIEREN. SIE DIENEN LEDIGLICH DAZU, DIE SYNTAX DES ERA-KOMMANDOS ZU VERSTEHEN !!)**

```
ERA ED.COM

ERA ED.COM,PROFILE.GER

ERA *.COM

ERA E???????.C?M
```

In allen Fällen würde **mindestens ED.COM gelöscht** werden.

E R A S E

Durch **ERASE variable,...** wird die vorherige Dimensionierung der **variable** durch ein **DIM**-Kommando rückgängig gemacht, so daß der ehemals belegte Speicherplatz für neue Zwecke nutzbar wird.
Es ist nicht möglich, eine bereits ge**ERASE**te Variable erneut zu löschen.

Beispiel:

```
10 DIM a(20,20),b(40)                  | Variablendfelder festlegen
20 FOR i=1 to 20                       |
30   FOR u=1 to 20                     | Variablen in Schleifen
40       a(i,u)=u                      |
50   NEXT                              | mit irgendwelchen Werten
60 NEXT                                |
70 FOR i=1 to 40                       | auffüllen
80   b(i)=40-i                         |
90 NEXT                                |
100 PRINT "a(10,15)=";a(10,15)         | und anzeigen
110 PRINT "b(19)=";b(19)
120 PRINT "Noch freier Speicherplatz vor ERASE:";FRE("");"Bytes"
130 ERASE a,b
140 PRINT "Freier Speicherplatz nach ERASE:";FRE("");"Bytes"
150 END
```

RUN:

```
a(10,15)= 15
b(19)= 21
Noch freier Speicherplatz vor ERASE: 29316 Bytes          | Normalwerte bei Grund-
Freier Speicherplatz nach ERASE: 31264 Bytes              | einstellung
Ok
■
```

E R L

Die Funktion **ERL** (**ER**ror **L**ine) wird dazu benutzt, festzustellen, in welcher Zeile der letzte Fehler auftrat. Es wird also die fehlerhafte Zeilennummer übergeben.

Wenn **ERL** in Vergleichsoperationen benutzt wird, muß es auf der linken Seite der zu vergleichenden Ausdrücke stehen, damit BASIC die rechte Seite als Zeilennummer erkennt, so daß es bei späterer Anwendung des **RENUM**-Befehles keine Probleme gibt und der Programmablauf nicht gestört wird.

Beispiel:

```
10 ON ERROR GOTO 100                          | Bei Fehler Zeile 100 anspringen
20 PRINT "Diese Zeile ist korrekt,"
30 PRINT "die nächste nicht."
40 PRINZ "Hier in Zeile 40 ist »PRINZ« ein SYNTAX-ERROR"
50 END
100 PRINT "In Zeile";ERL;"tritt der Fehler mit der Nummer";ERR;"auf."
110 RESUME NEXT
```

RUN:

```
Diese Zeile ist korrekt,
die nächste nicht.
In Zeile 40 tritt der Fehler mit der Nummer 2 auf.
Ok
■
```

E R R

Die Funktion **ERR** wird dazu benutzt, festzustellen, welcher Fehler zuletzt auftrat. **ERR** liefert den entsprechenden Fehlercode. Eine ausführliche Beschreibung finden Sie im Anhang II Ihres mitgelieferten BASIC-Benutzer-Handbuches.

Beispiel:

```
10 ON ERROR GOTO 100                          | bei Fehler Zeile 100 anspringen
20 PRINT "Diese Zeile ist korrekt,"
30 PRINT "die nächste nicht."
40 PRINZ "Hier in Zeile 40 ist »PRINZ« ein SYNTAX-ERROR"
50 END
100 PRINT "In Zeile";ERL;"tritt der Fehler mit der Nummer";ERR;"auf."
110 RESUME NEXT
```

RUN:

```
Diese Zeile ist korrekt,
die nächste nicht.
In Zeile 40 tritt der Fehler mit der Nummer 2 auf.
Ok
■
```

E R R O R

ERROR fehlernummer simuliert den mit **fehlernummer** spezifizierten Fehler. **fehlernummer** stellt eine Integerzahl im Bereich von 1 bis 255 dar. Eine ausführliche Beschreibung der Fehlercodes finden Sie im Anhang II Ihres BASIC-Handbuches.

Beispiel:

```
10 ON ERROR GOTO 100                          | bei Fehler Zeile 100 anspringen
20 PRINT "Diese Zeile ist korrekt,"
30 PRINT "in der nächsten wird ein SYNTAX-ERROR simuliert."
40 ERROR 2                                    | 2 = SYNTAX ERROR
50 END
100 PRINT "In Zeile";ERL;"tritt der Fehler mit der Nummer";ERR;"auf."
110 RESUME NEXT
```

RUN:

```
Diese Zeile ist korrekt,
in der nächsten wird ein SYNTAX-ERROR simuliert.
In Zeile 40 tritt der Fehler mit der Nummer 2 auf.
```

E X P

Die Funktion **EXP (p)** erhebt die Zahl e in die p-te Potenz. e (= 2.718281828) ist diejenige Zahl, deren natürlicher Logarithmus 1 ist.
p darf dabei nicht größer als +88.0296... und nicht kleiner als -88.7228... sein, da **EXP** sonst entweder einen Überlauffehler erzeugt, oder 0 als Ergebnis errechnet.
Die Funktion **EXP** liefert nur einfach genaue Zahlen.

Beispiel:

```
10 PRINT "EXP (8) ergibt:";EXP (8)
20 END
```

RUN:

```
EXP (8) ergibt: 2980.958
Ok
■
```

F E T C H K E Y $ - F I E L D

Bei diesen Kommandos handelt es sich um **JETSAM**-Befehle. Genaueres erfahren Sie im **JETSAM**-Teil auf den Seiten 173 und 178.

F I L E S

FILES datei$** listet die mit **datei$ spezifizierte(n) Dateie(n) auf dem Bildschirm auf. In **datei$** dürfen auch auf übliche Weise Wildcards (?,*) verwendet werden.

Beispiel:

```
FILES "BASIC.COM"

FILES "*.COM"

FILES "B???????.C?M"
```



```
10 datei$="BASIC.COM"
20 FILES datei$
30 END
```

Wenn sich BASIC.COM auf der Diskette befindet, sollte in allen Fällen (ggf. nach **RUN**) mindestens

```
BASIC.COM
Ok
■
```

auf dem Bildschirm erscheinen.

F I N D $

Die Funktion **FIND$(maske$,n-te_Datei)** sucht nach der **n-ten_Datei**, auf die die durch **maske$** spezifizierte Maske zutrifft. Falls sie eine solche findet, übergibt sie einen 12 Zeichen langen String mit der gefundenen Datei, in dem auch evtl. Bit 7 gesetzt sein kann. Andernfalls wird ein leerer String übergeben.

Bit 7 gesetzt in

Byte 10:	Read Only-Attribut
Byte 11:	System-Attribut
Byte 12:	Archive-Attribut

In **maske$** dürfen auch auf übliche Weise Wildcards (?,*) verwendet werden.

n-te_Datei muß eine Integerzahl zwischen 1 und 255 ergeben. Falls es weggelassen wird, sucht **FIND$** nach der ersten zutreffenden Datei.

Beispiel:

```
10 maske$="*.COM"
20 n=1
30 datei$=FIND$ (maske$,n)
40 IF datei$<>"" THEN PRINT datei$:n=n+1:GOTO 30
50 PRINT "Gefundene Dateien:";n
60 END
```

RUN:

```
Hier wird die erste .COM-Datei gezeigt.
          . . . . . .
Hier wird die n-te .COM-Datei gezeigt.
Gefundene Dateien: n
Ok
■
```

F I X

Die Funktion **FIX (zahl)** rundet **zahl** auf eine ganze Zahl in Richtung 0 ab, es werden also die Nachkommastellen von **zahl** abgeschnitten.

Beispiel:

```
10 a=12.34567
20 b=-123456.7
30 PRINT "FIX (";a;") =";FIX(a)
40 PRINT "FIX (";b;") =";FIX(b)
50 END
```

RUN:

```
FIX ( 12.34567 ) = 12
FIX (-123456.7 ) =-123456
Ok
■
```

F O R

Mit **FOR variable=startwert TO endwert STEP schrittweite** lassen sich ohne große Probleme Schleifen realisieren.

variable ist eine Kontrollvariable für den aktuellen Zählerstand der Schleife. Sie darf nur eine ganze oder einfach genaue Variable sein.

startwert stellt den Anfangswert der Schleife dar.

endwert stellt den Endwert der Schleife dar. Die Schleife wird genau dann beendet, wenn **variable** dem Wert nach den **endwert** erreicht oder beim nächsten Durchgang durch ein zu großes Inkrement überschreiten würde.

schrittweite stellt ein Inkrement dar, um welches **variable** bei jedem Schleifendurchgang erhöht bzw. erniedrigt wird. Ist es nicht angegeben wird +1 angenommen.

Ist der **startwert** bei einer positiven **schrittweite** größer als der **endwert**, wird die Schleife übersprungen. Dasselbe gilt auch für den Fall, daß der **startwert** bei einer negativen **schrittweite** kleiner als der **endwert** ist.
Zu beachten ist, daß jede **FOR**-Schleife durch ein **NEXT**-Kommando beendet werden muß.

Beispiel:

```
10 DIM a(20,20),b(40)
20 FOR i=1 to 20
30   FOR u=20 to 1 STEP -1
40       a(i,u)=u
50   NEXT
60 NEXT
70 FOR i=1 to 40
80   b(i)=40-i
90 NEXT
100 PRINT "a(10,15)=";a(10,15)
110 PRINT "b(19)=";b(19)
120 END
```

RUN:

```
a(10,15)= 15
b(19)= 21
Ok
■
```

F R E

FRE ("") bestimmt den noch zur freien Benutzung verfügbaren Speicherplatz, wobei eine "garbage collection" (Speicheraufräumung) durchgeführt wird. Soll sie nicht bei der Speicherplatzbestimmung durchgeführt werden, muß **FRE(0)** verwendet werden..

Beispiel:

```
10 DIM a(20,20),b(40),c(500),a$(20)        | ordentlich Speicherplatz verbrauchen...
20 FOR i=1 to 20
30   a$(i)=STRING$(255,"T")
40 NEXT
50 PRINT "Noch freier Speicherplatz vor ERASE:";FRE("");"Bytes"
60 ERASE a,b,c,a$
70 PRINT "Freier Speicherplatz nach ERASE:";FRE("");"Bytes"
80 END
```

RUN:

```
Noch freier Speicherplatz vor ERASE: 22171 Bytes
Freier Speicherplatz nach ERASE: 31344 Bytes
Ok
■
```

G E T

Bei diesem Kommando handelt es sich um einen Befehl, um in Jetsam-Dateien bzw. Dateien mit wahlfreiem Zugriff Datensätze einzulesen. Genaueres erfahren Sie im Teil über **JETSAM** bzw. **Dateien mit wahlfreiem Zugriff** auf den Seiten 189 und 206.

G O S U B

Das **GOSUB zeilennummer**-Kommando wird dazu benutzt, um aus dem laufenden Programm ein Unterprogramm aufzurufen und dann (nach Beendigung des Unterprogramms durch **RETURN**) mit dem unmittelbar auf **GOSUB** folgenden Befehl fortzufahren.

Beispiel:

```
10 PRINT "Hier läuft das Hauptprogramm."
20 PRINT "Gleich wird das Unterprogramm aufgerufen."
30 GOSUB 100
40 PRINT "Jetzt läuft das Hauptprogramm weiter."
50 END
100 PRINT CHR$(7);"---- UNTERPROGRAMM ----";CHR$(7)
110 RETURN
```

RUN:

```
Hier läuft das Hauptprogramm.
Gleich wird das Unterprogramm aufgerufen.
(Pieps)---- UNTERPROGRAMM ----(Pieps)
Jetzt läuft das Hauptprogramm weiter.
Ok
■
```


G O T O

Das **GOTO zeilennummer**-Kommando stellt einen unbedingten Sprung zur durch die Zahl **zeilennummer** angegebenen Zeile dar.

Beispiel:

```
10 PRINT "Hier läuft das Programm."
20 PRINT "Jetzt springt es zur Zeile 100."
30 GOTO 100
40 END
100 PRINT CHR$(7);"Dies ist Zeile 100.";CHR$(7)
110 GOTO 40
```

RUN:

```
Hier läuft das Programm.
Jetzt springt es zur Zeile 100.
(Pieps)Dies ist Zeile 100.(Pieps)
Ok
■
```

H E X $

Die Funktion **HEX$ (zahl,stellen)** dient dazu, **zahl** in einen hexadezimalen String umzuwandeln, wobei **stellen** die Länge des hexadezimalen Stringes angibt. Falls die Länge unterschritten wird, wird der String bis zur vorgegebenen Länge mit führenden Nullen aufgefüllt. Ist der String jedoch zu lang, wird nichts abgeschnitten.
Zu beachten ist, daß **zahl** nur zwischen 0 und 65535 und **stellen** nur zwischen 0 und 16 liegen darf.

Beispiel:

```
10 a=256
20 b=65535
30 PRINT "HEX$(";a;") ergibt ";HEX$(a,4)
40 PRINT "HEX$(";b;") ergibt ";HEX$(b,4)
50 END
```

RUN:

```
HEX$( 256 ) ergibt 0100
HEX$( 65535 ) ergibt FFFF
Ok
■
```

H I M E M

Die **HIMEM**-Funktion ermittelt die höchste für BASIC benutzbare Speicheradresse. Sie bezeichnet den Bereich, in dem BASIC Speicherplatz für zahlreiche Zwecke reservieren kann, z.B. Schleifen, Variablen (insbesondere String-Variablen), etc.

Beispiel:

```
10 PRINT HEX$(HIMEM)
20 END
```

RUN:

```
F605 (in der Regel)
Ok
■
```

I F

Das **IF**-Kommando ermöglicht bedingt auszuführende Befehle.
Die Form ist grundsätzlich: **IF vergleichsoperation THEN anweisung ELSE anweisung**. Wenn das Ergebnis der **vergleichsoperation** wahr ist, wird die mit **THEN** verbundene **anweisung** ausgeführt. Ist das Ergebnis unwahr, wird die mit **ELSE** (sofern vorhanden) verbundene **anweisung** ausgeführt. Als Vergleichsoperator kann folgendes eingesetzt werden: = (gleich), < (kleiner), > (größer), <> (ungleich), **AND**, **NOT**, **OR**, **XOR** (letzteres bezieht sich auf eine bitweise Verknüpfung).
Die **anweisung** besteht aus ein oder mehreren Befehlen, die voneinander durch Doppelpunkte getrennt werden. Sie kann aber auch aus einer Zeilennummer bestehen: die Formen **THEN zeilennummer** und **ELSE zeilennummer** sind gleichbedeutend mit **THEN GOTO zeilennummer** und **ELSE GOTO zeilennummer**. Statt **THEN zeilennummer** kann aber auch **GOTO zeilennummer** verwendet werden.
Zu beachten ist, daß der Wirkungsbereich des **IF**-Kommandos erst durch das Zeilenende beendet wird. Es ist also in einer Zeile nicht möglich, nach einem **IF**-Kommando unabhängig von diesem arbeitende Befehle zu verwenden.

Beispiel:

```
10 INPUT "Ihr Name: ",name$
20 PRINT "Herr / Frau ([H] / [F])";
30 t$=UPPER$(INPUT$(1))
40 IF t$<>"H" AND t$<>"F" THEN 30
50 PRINT t$:PRINT
60 PRINT "Guten Tag, ";
70 IF t$="H" THEN PRINT "Herr "; ELSE PRINT "Frau";
80 PRINT name$
90 END
```

RUN:

```
Ihr Name: Mustermann
Herr / Frau ([H] / [F])? H

Guten Tag, Herr Mustermann
Ok
■
```

I N K E Y $

Mit **INKEY$** kann abgefragt werden, ob ein Zeichen von der Tastatur anliegt. Ist dies der Fall, wird es durch **INKEY$** übergeben. Ist dies nicht der Fall, wird ein leerer String übergeben. Sofern **OPTION RUN** aktiv ist, werden alle Zeichen übergeben, die von der Tastatur kommen.Ist es nicht aktiv, rufen ↑C ([ALT]+[C]) und ↑S ([ALT]+[S]) ihre üblichen Funktionen hervor (↑C bricht das Programm ab, ↑S läßt das Programm bis zum nächsten Tastendruck warten).

Beispiel:

```
10 PRINT "Abbruch mit ↑C (STOP)"
20 t$=INKEY$
30 WHILE t$=""
40 t$=INKEY$
50 WEND
60 PRINT "Es wurde die Taste mit dem ASCII-Code";ASC(t$);"gedrückt."
70 GOTO 20
```

RUN:

```
Abbruch mit †C (Stop)
Es wurde die Taste mit dem ASCII-Code xx gedrückt.
Es wurde die Taste mit dem ASCII-Code yy gedrückt.
†C            | Sie haben die STOP-Taste gedrückt
Break in 40
Ok
■
```

I N P

Die **INP (kanalnummer)**-Funktion dient dazu, aus dem durch **kanalnummer** (0..255) spezifizierten Kanal des I/O-Portes einen Wert (0..255) zu lesen.

Beispiel:

```
10 INPUT "Kanalnummer (0..255): ",kanal
20 IF kanal<0 OR kanal>255 THEN 10
30 PRINT "Am Kanal";kanal;"des I/O-Portes liegt der Wert";INP (kanal);"an."
40 END
```

RUN:

```
Kanalnummer (0..255): xxx
Am Kanal xxx des I/O-Portes liegt der Wert yyy an.
Ok
■
```

I N P U T

INPUT ;"Text in Anführungszeichen";variablenliste dient zur Dateneingabe von der Konsole (meistens die Tastatur), wobei auf dem Bildschirm die eingegebenen Daten erscheinen und editiert werden können.

Das erste Semikolon bewirkt, daß der Cursor nach Beendigung der Eingabe nicht in die nächste Zeile springt. Es kann auch weggelassen werden.

Wenn ein Text (der von Anführungszeichen eingeschlossen sein muß) angegeben wird, wird er vor der Dateneingabe auf dem Bildschirm ausgegeben. Unmittelbar darauf folgt dann die Dateneingabe.

Steht vor der **variablenliste** ein Semikolon, wird vor der

Dateneingabe ein Fragezeichen ausgegeben. Steht statt des Semikolons ein Komma, entfällt das Fragezeichen.

Die Variablen der **variablenliste** enthalten nach Beendigung der Eingabe die einzelnen eingegebenen Werte. **variablenliste** ist in der Form **variable1,variable2,variable3,..** anzugeben. Selbstverständlich kann die **variablenliste** auch nur aus einer einzigen Variable bestehen.

Zu beachten ist, daß mit der Eingabe eines Kommas auch ein Element folgen muß. Wenn mehr Elemente eingegeben werden, als entsprechende Variablen vorhanden sind, oder wenn der Datentyp des eingegebenen Elementes nicht mit dem der Variablen übereinstimmt, wird die Fehlermeldung **?Redo from start** ausgegeben und das **INPUT**-Kommando erneut gestartet.

Die Dateneingabe wird durch einen Wagenrücklauf ([RETURN] / [ENTER]) abgeschlossen.

Beispiel:

```
INPUT a$
?■

INPUT "Text ";a$
Text ?■

INPUT "Text ",a$,b$,c
Text Test1,Test2,800■

INPUT ;"Zahl ",b:PRINT " Hier erfolgte kein Carriage Return !"
Zahl xx■ Hier erfolgte kein Carriage Return !
```

I N P U T

INPUT #dateinummer,variablenliste wird zum Lesen von Daten aus sequentiellen Dateien benutzt.
dateinummer nennt die Datei, aus der die Daten gelesen werden sollen.
variablenliste stellt eine Liste von Variablen dar, die festlegt, in welche Variablen die Daten gelesen werden sollen. Sie ist in der Form **variable1,variable2,variable3,..** anzugeben. Dabei geben die Variablen den Datentyp der zu lesenden Variablen an.

Die drei Datentypen:

numerischer Wert: Ein numerisches Element wird beendet durch Leerstellen, Komma, Wagenrücklauf oder Dateiende.

String mit "": Alle Zeichen zwischen dem führenden und dem abschließenden Anführungszeichen bilden den String, in dem auch Wagenrücklauf oder Zeilenvorschub eingeschlossen sein

dürfen. Der String wird durch Anführungszeichen oder das Dateiende abgeschlossen.

einf. String: Der einfache String beginnt bei dem eigentlichen Text (führende Leerzeichen werden ignoriert) und endet durch Komma, Wagenrücklauf oder Dateiende. Strings sind auf 255 Zeichen begrenzt und werden nach dem 255. Zeichen automatisch abgeschlossen.

Beispiel:

```
10 DIM inhalt$(10)
20 satz=0
30 OPEN "I",#1,"RPED.SUB"
40 WHILE NOT EOF(1)
50   satz=satz+1
60   INPUT #1,inhalt$(satz)
70 WEND
80 CLOSE
90 FOR i=1 to satz
100   PRINT "Satz ";DEC$(i,"##");" = ";inhalt$(i)
110 NEXT
120 END
```

RUN:

```
Satz 01 = basic rped
Ok
■
```

I N P U T $

INPUT$ (max_länge,#dateinummer) wird zur Stringeingabe mit fester Länge von der Konsole (meistens Tastatur) oder aus einer Datei benutzt.

Die **max_länge** bezeichnet die Länge des zu lesenden Strings, die zwischen 1 und 255 liegen muß.

Die **dateinummer** bezeichnet die Nummer der Datei, aus der gelesen werden soll. Wird sie nicht angegeben, liest **INPUT$** den String von der Konsole. Sofern **OPTION RUN** aktiv ist, werden alle Zeichen, die von der Tastatur kommen, aufgenommen. Ist es nicht aktiv, rufen ↑C und ↑S ihre üblichen Funktionen hervor (↑C bricht das Programm ab, ↑S läßt das Programm bis zum nächsten Tastendruck warten).

Beispiel:

```
10 DIM zeichen$(100)
20 OPEN "I",#1,"PROFILE.GER"
30 FOR zeichen=1 to 100                    | Hier wird die Datei
40   zeichen$(zeichen)=INPUT$ (1,#1)       | zeichenweise eingelesen
50 NEXT
60 CLOSE
70 FOR i=1 to 100
80   PRINT zeichen$(i);
90 NEXT
100 END
```

RUN:

```
setdef m:,* [order = (sub,com) temporary = m:]
pip
<m:=basic.com[o]
<m:=dir.com[o]
<m:=erase.com[o]
Ok
■
```

I N S T R

Die **INSTR (begin,zu_durchsuchender_String,gesuchter_String)**-Funktion wird dazu benutzt, ab der mit **begin** genannten Stelle im **zu_durchsuchenden_String** nach dem **gesuchten_String** zu suchen.

Wird **begin** (1..255) weggelassen, beginnt die Suche nach **gesuchter_String** beim ersten Zeichen des **zu_durchsuchenden_Strings.**

Wird der gesuchte String gefunden, übergibt **INSTR** die Position des ersten Auftretens des gesuchten Strings. Ist der gesuchte String nicht enthalten, wird der Wert 0 übergeben.

Sind im **zu_durchsuchenden_String** ab **begin** keine Zeichen mehr vorhanden, wird ebenfalls der Wert 0 übergeben. Wenn der **gesuchte_String** ein leerer ist, wird er sofort gefunden, falls der zuletzt genannte Fall nicht eintritt.

Beispiel:

```
10 begriff$="Bergstiefel"
20 such$="stiefel"
30 posi=INSTR(begriff$,such$)
40 PRINT "Das Wort ";such$;" befindet sich in ";begriff$;" an";posi;"ter Stelle."
50 END
```

RUN:

```
Das Wort stiefel befindet sich in Bergstiefel an 5 ter Stelle.
Ok
■
```

I N T

INT (zahl) rundet **zahl** nach unten zur nächsten ganzen Zahl ab.

Beispiel:

```
10 a=12.3456
20 b=-12.3456
30 PRINT "INT (";a;") ergibt ";INT(a)
40 PRINT "INT (";b;") ergibt ";INT(b)
```

RUN:

```
INT ( 12.3456 ) ergibt 12
INT (-12.3456 ) ergibt -13
Ok
■
```


K I L L

KILL dateiname$** löscht die mit **dateiname$ spezifizierte Datei, wobei auch Wildcards (?,*) auf übliche Weise verwendet werden dürfen. Hiermit ist jedoch Vorsicht geboten, da keine Sicherheitsabfrage wie unter CP/M Plus stattfindet.

Beispiel: **(ACHTUNG: ES IST SINNVOLL, DIE BEISPIELE NICHT AUSZUPROBIEREN. SIE DIENEN LEDIGLICH DAZU, DIE SYNTAX DES KILL-KOMMANDOS ZU VERSTEHEN !!)**

```
KILL "ED.COM"

datei$="ED.COM":KILL datei$

datei$="*.COM":KILL datei$

KILL "E???????.C?M"
```

In allen Fällen würde **mindestens** ED.COM gelöscht werden.

L E F T $

LEFT$ (wort$,n) ermittelt von **wort$** die ersten **n** linken Zeichen. **n** muß dabei zwischen 0 und 255 liegen.

wort$ stellt einen beliebigen String dar, der zwischen 1 und 255 Zeichen lang sein muß.

Ist **n** größer als die gesamte Länge von **wort$**, wird ein String übergeben, der aus **wort$** und der Anzahl Leerstellen besteht, die der Länge von **wort$** bis zur Länge **n** fehlen.

Beispiel:

```
10 text$="JOYCE - mehr als ein Textsystem."
20 links$=LEFT$(text$,5)
30 PRINT links$
40 END
```

RUN:

```
JOYCE
Ok
■
```

L E N

LEN (text$) ermittelt die gesamte Länge (0..255) von **text$**, eingeschlossen sind auch Zeichen, die nicht ohne weiteres auf dem Bildschirm darstellbar sind (Zeichen, deren ASCII-Code kleiner als 32 ist).

Liefert **LEN** als Ergebnis 0, zeigt das, daß **text$** ein leerer String ist, also keine Zeichen enthält.

Beispiel:

```
10 text$="JOYCE"
20 PRINT "Die Länge des Wortes ";text$;" beträgt";LEN (text$);" Zeichen"
30 END
```

RUN:

```
Die Länge des Wortes JOYCE beträgt 5 Zeichen
Ok
■
```

L E T

Das **LET**-Kommando wurde früher zur Wertzuweisung für Variablen benutzt. In Mallard 80-BASIC hat es jedoch keine besondere Funktion und kann daher ignoriert werden.

L I N E I N P U T

LINE INPUT ;"Text in Anführungszeichen";variable$ dient zur Dateneingabe von der Konsole (meistens die Tastatur), wobei auf dem Bildschirm die eingegebenen Daten erscheinen und editiert werden können. **LINE INPUT** übernimmt vollständig die eingegebene Zeile, wobei also auch z.B. Kommas mitgelesen werden. Dadurch ist es jedoch nicht möglich, mit einem **LINE INPUT**-Kommando mehreren Variablen verschiedene Texte zuzuweisen. Auch kann aus diesen technischen Gründen immer nur ein **Text** eingegeben werden, der dann einer Stringvariable zugewiesen wird.

Das erste Semikolon bewirkt, daß der Cursor nach Beendigung der Eingabe nicht in die nächste Zeile springt. Es kann auch bei Bedarf weggelassen werden, damit ein Carriage Return

erfolgt.

Wenn ein Text (der von Anführungszeichen eingeschlossen sein muß) angegeben wird, wird er vor der Dateneingabe auf dem Bildschirm ausgegeben. Unmittelbar darauf folgt dann die Dateneingabe.

Steht vor **variable$** ein Semikolon, wird vor der Dateneingabe ein Fragezeichen ausgegeben. Steht statt des Semikolons ein Komma, entfällt das Fragezeichen.

variable$ enthält nach Beendigung der Eingabe, die durch einen Wagenrücklauf ([RETURN] / [ENTER] / [ALT]+[M]) abgeschlossen wird, den eingegebenen Text.

Beispiel:

```
LINE INPUT a$
?■

LINE INPUT "Text ";a$
Text ?■

LINE INPUT ;"Zahl ",b:PRINT " Hier erfolgte kein Carriage Return !"
Zahl xx■ Hier erfolgte kein Carriage Return !
```

L I N E I N P U T

LINE INPUT #dateinummer,text$ wird zum Lesen von Daten aus einer Datei benutzt.

dateinummer nennt die Datei, aus der die Daten gelesen werden sollen.

text$ stellt die Variable dar, in die die Daten gelesen werden sollen. Die Daten, die immer vom Typ "String" sind, enden entweder mit einem Wagenrücklauf (=Zeilenende) oder automatisch mit dem 255. Zeichen.

Beispiel:

```
10 DIM inhalt$(50)                                  | Initialisierung
20 satz=0                                           |
30 OPEN "I",#1,"PROFILE.GER"                        |
40 WHILE NOT EOF(1)                                 | Leseroutine
50   satz=satz+1                                    | Datensatzzähler
60   LINE INPUT #1,inhalt$(satz)                    | Auslesen der Datei
70 WEND                                             | Ende Routine
80 CLOSE                                            | Datei schließen
90 FOR i=1 to satz                                  | Schleife zum
100   PRINT "Satz ";DEC$(i,"##");" = ";inhalt$(i)   | Auslesen der
110 NEXT                                            | einzelnen Sätze
120 END                                             | Programmende
```

RUN:

```
Satz  1 = setdef m:,* [order = (sub,com) temporary = m:]
Satz  2 = pip
Satz  3 = <m:=basic.com[o]
Satz  4 = <m:=dir.com[o]
Satz  5 = <m:=erase.com[o]
Satz  6 = <m:=paper.com[o]
Satz  7 = <m:=pip.com[o]
Satz  8 = <m:=rename.com[o]
Satz  9 = <m:=setkeys.com[o]
Satz 10 = <m:=show.com[o]
Satz 11 = <m:=submit.com[o]
Satz 12 = <m:=type.com[o]
Satz 13 = <
Ok
■
```

L I S T

Durch das Kommando **LIST zeilennummernbereich** lassen sich die mit **zeilennummernbereich** festgelegten Zeilen aus dem aktuellen Programm auf der Konsole (meistens Bildschirm) auflisten.

zeilennummernbereich spezifiziert alle Zeilen, deren Nummern im gegebenen Bereich liegen. Dieser Bereich kann eines der folgenden Formate annehmen:

zeilennummer	nur diese Nummer fällt in den Bereich
zeilennummer-zeilennummer	Zeilen von der ersten Nummer bis einschließlich der letzten
zeilennummer-	Zeilen von der bestimmten Zeile bis zum Ende des Programms
-zeilennummer	Zeilen vom Programmanfang bis zu der bestimmten Zeile

Beispiel:

```
LIST 40          | Auflisten der Zeile 40

LIST 10-30       | Auflisten der Zeilen 10 bis 30

LIST 100-        | Auflisten des Programms ab Zeile 100

LIST -100        | Auflisten des Programms bis Zeile 100
```


L L I S T

Durch das Kommando **LLIST zeilennummernbereich** lassen sich die mit **zeilennummernbereich** festgelegten Zeilen aus dem aktuellen Programm auf dem LST-Kanal (meistens Drucker) ausgeben.
zeilennummernbereich spezifiziert alle Zeilen, deren Nummern im gegebenen Bereich liegen. Dieser Bereich kann eines der folgenden Formate annehmen:

zeilennummer	nur diese Nummer fällt in den Bereich
zeilennummer-zeilennummer	Zeilen von der ersten Nummer bis einschließlich der letzten
zeilennummer-	Zeilen von der bestimmten Zeile bis zum Ende des Programms
-zeilennummer	Zeilen vom Programmanfang bis zu der bestimmten Zeile

Beispiel:

```
LLIST 40         | Ausdrucken der Zeile 40

LLIST 10-30      | Ausdrucken der Zeilen 10 bis 30

LLIST 100-       | Ausdrucken des Programms ab Zeile 100

LLIST -100       | Ausdrucken des Programms bis Zeile 100
```

L O A D

Mit **LOAD datei$,R** läßt sich eine Datei von der Diskette laden und bei der Angabe des **,R** (Run)-Parameters sofort starten. Ein evtl. schon im Speicher befindliches Programm wird überschrieben (= gelöscht!).

datei$** gibt den Namen der Datei, die geladen werden soll. Es kann dabei am Anfang von **datei$ auch eine Laufwerksbezeichnung mit angegeben werden. Wird keine Extension mit angegeben, wird **.BAS** angenommen.

Zu beachten ist, daß alle Variablen und Benutzerfunktionen gelöscht, alle **DEFINT-**, **DEFSNG-**, **DEFDBL-**, **DEFSTR-** und **OPTION BASE**-Einstellungen zurückgesetzt werden.

Beispiel:

```
LOAD "RPED

LOAD "RPED.BAS

LOAD "A:RPED

LOAD "A:RPED.BAS

LOAD "RPED.BAS",R
```

In sämtlichen Fällen wird versucht, **RPED.BAS** zu laden.

L O C - L O C K

Bei diesen Kommandos handelt es sich um Befehle, die bei der Verarbeitung von Jetsam-Dateien bzw. Dateien mit wahlfreiem Zugriff verwendet werden. Genaueres erfahren Sie im Teil über **JETSAM** bzw. **Dateien mit wahlfreiem Zugriff** auf den Seiten 194 und 208.

L O F

LOF (dateinummer) wird zum Ermitteln der Länge (Length Of File) der mit **dateinummer** gekennzeichneten Datei benutzt. Der übergebene Wert bezieht sich dabei auf die Dateilänge in Records (1 Record = 128 Bytes).

ACHTUNG: Die **LOF**-Funktion arbeitet nur bis zu einer maximalen Dateilänge von 128 Records (=16384 Bytes = 16 Kbyte) korrekt. Eine größere Datei wird konstant mit 128 Records angegeben. Der Grund für diese Fehlfunktion liegt im Aufbau des Directory-Eintrags einer Datei auf der Diskette. Für jede Datei, die größer als 128 Records ist, wird für jede angefangenen 128 Records ein neuer Directory-Eintrag angelegt. **LOF** ermittelt jedoch immer nur die Länge des gerade aktuellen Directory-Eintrages, der für die Daten zuständig ist, die man liest. Wenn man jetzt also die gesamte Länge einer Datei benötigt, muß man die einzelnen Längen der Directory-Einträge addieren.

Beispiel:

```
10 OPEN "I",#1,"RPED.BAS"
20 PRINT "Die Länge der Datei beträgt";LOF(1);"Records =";LOF(1)*128;"Bytes."
30 CLOSE
40 END
```


RUN:

```
Die Länge der Datei beträgt 56 Records = 7168 Bytes.
OK
■
```

L O G

LOG (zahl) berechnet den natürlichen Logarithmus von **zahl**. **LOG** liefert einen einfach genauen Wert, da **zahl**, die größer als 0 sein muß, vor der Berechnung des Logarithmus in eine einfach genaue Zahl umgewandelt wird.

Beispiel:

```
10 PRINT "LOG (148.4132) ergibt:";LOG (148.4132)
20 END
```

RUN:

```
LOG (148.4132) ergibt: 5
Ok
■
```

L O G 1 0

LOG10 (zahl) berechnet den dekadischen Logarithmus (zur Basis 10) von **zahl**. **LOG10** liefert einen einfach genauen Wert, da **zahl**, die größer als 0 sein muß, vor der Berechnung des Logarithmus in eine einfach genaue Zahl umgewandelt wird.

Beispiel:

```
10 PRINT "LOG10 (100) ergibt:";LOG10 (100)
20 END
```

RUN:

```
LOG10 (100) ergibt: 2
Ok
■
```

L O W E R $

LOWER$ (text$) wandelt alle Großbuchstaben in **text$** in Kleinbuchstaben um. Als Großbuchstaben werden nur die Zeichen mit dem ASCII-Code zwischen 65 und 90 (A-Z) anerkannt und umgewandelt.

Beispiel:

```
10 text$="JOYCE"
20 PRINT text$;" in Kleinbuchstaben umgewandelt ergibt ";LOWER$(text$)
30 END
```

RUN:

```
JOYCE in Kleinbuchstaben umgewandelt ergibt joyce
OK
■
```

L P O S (0)

LPOS(0) ermittelt die aktuelle logische Druckposition des Druckers. Das heißt, daß die ermittelte Position nicht unbedingt etwas mit der tatsächlichen zu tun hat, weil jedes Zeichen, dessen ASCII-Code kleiner als 32 ist, nicht mitgezählt wird. Eine Ausnahme machen nur die Zeichen mit dem ASCII-Code 8, 9 und 13.

Die Bedeutung:

ASCII:
- 8 entspricht einem Rückschritt, der den Zähler um 1 verringert, falls die Position nicht schon 1 ist.
- 9 entspricht einem TAB, der in Leerstellen umgewandelt wird, von denen jede gezählt wird
- 13 entspricht einem Wagenrücklauf, der die Position auf 1 setzt.

Beispiel:

```
10 LPRINT "Dies ist ein Test.";
20 PRINT "Die aktuelle Druckposition beträgt:";LPOS(0)
30 END
```


RUN:

```
Dies ist ein Test. (Erscheint auf dem Drucker)
Die aktuelle Druckposition beträgt 19
Ok
■
```

L P R I N T

LPRINT text; bewirkt die Ausgabe des **textes** auf dem Drucker.

text kann dabei eine Variablenliste oder ein fester Ausdruck sein. Die Variablenliste kann aus Variablen vom Typ "string" und "numerisch" bestehen. Wenn die einzelnen Variablen dabei durch ein Komma getrennt werden, wird jedesmal für ein Komma ein Tabulator eingefügt. Wenn die Variablen durch ein Semikolon getrennt werden, werden die Werte der Variablen ohne zusätzliche Leerstellen aneinandergereiht gedruckt. In diesem Fall ist jedoch zu beachten, daß eine Variable numerischen Typs eine Leerstelle vor (für Vorzeichen) und nach der Zahl beansprucht. Der feste Ausdruck ist entweder eine Zahl oder ein Text in Anführungsstrichen.

Ein Semikolon am Ende des Textes bewirkt, daß kein Wagenrücklauf ausgeführt wird. Ein Komma würde einem Tabulator gleichkommen, wobei auch hier kein Wagenrücklauf ausgeführt wird. Wenn am Ende kein Zeichen steht, wird ein ganz normaler Zeilenvorschub ausgeführt.

Beispiel:

```
10 a=8256
20 a$="SCHNEIDER JOYCE PCW"
30 LPRINT "Dieses Beispiel läuft auf einem ";a$;a
40 a=10:b=20:c=-30
50 LPRINT a,b,c,999;
60 LPRINT "Test"
70 END
```

RUN:

(Erscheint auf dem Drucker):

```
Dieses Beispiel läuft auf einem SCHNEIDER JOYCE PCW 8256
 10            20           -30            999 Test
```

L S E T

Bei diesem Kommando handelt es sich um einen **JETSAM**-Befehl. Genaueres erfahren Sie im **JETSAM**-Teil auf Seite 173.

M A X

MAX (zahlenliste) ermittelt den größten in **zahlenliste** enthaltenen Wert. **zahlenliste** ist in der Form **zahl1,zahl2, zahl3,..** anzugeben.

Beispiel:

```
10 a=10:b=20:c=-30
20 PRINT "MAX (";a;",";b;",";c;") ist";MAX (a,b,c)
30 END
```

RUN:

```
MAX ( 10 , 20 ,-30 ) ist 20
Ok
■
```

M E M O R Y

MEMORY High-Memory,Stack-Größe,Dateizahl,maximale_Satzlänge verändert die angegebenen Parameter.

High-Memory bezieht sich auf die obere vom BASIC benutzbare Speicheradresse (Sie bezeichnet den Bereich, in dem BASIC Speicherplatz für zahlreiche Zwecke reservieren kann, z.B. Schleifen, Variablen (insbesondere String-Variablen), etc.). Wird diese Angabe weggelassen, bleibt der aktuelle Wert eingestellt.

Stack-Größe gibt die Anzahl der von BASIC frei für das Stapelregister verwendbaren Bytes an. Der Stack ist der Speicher, der von BASIC für die Verwaltung von Schleifen, **GOSUB**-Routinen, etc. benutzt wird. Falls **Stack-Größe** angegeben wird, muß mindestens ein Wert von 256 verwendet werden. Auch werden dabei alle aktiven **FOR-**, **WHILE-**, und **GOSUB**-Kommandos vergessen. Wird diese Angabe weggelassen, bleibt der aktuelle Wert eingestellt.

Dateizahl gibt die maximale Anzahl gleichzeitig zu bearbeitender Dateien an. Ohne diese Angabe bleibt der aktuelle Wert eingestellt.

maximale_Satzlänge gibt die maximale Größe eines frei adressierbaren Satzes an. Ohne diese Angabe bleibt der aktuelle Wert eingestellt.

Die Standardeinstellungen:

High-Memory:	&HF605
Stack-Größe:	512 Bytes
Dateizahl:	3
max._Satzlänge:	128 Bytes

Beispiel:

```
MEMORY &HF605,,6          | High-Memory auf &HF605, maximale Dateizahl 6

MEMORY &HEFFF,512,3,256   | High-Memory auf &HEFFF, Stack-Größe auf 512 Bytes, 3 Dateien,
                          | maximale Satzlänge 128 Bytes

MEMORY ,,10               | 10 Dateien


10 PRINT "Alte obere Speicheradresse: ";HEX$(HIMEM)
20 MEMORY &HF5FF
30 PRINT "Neue obere Speicheradresse: ";HEX$(HIMEM)
40 END
```

RUN:

```
Alte obere Speicheradresse: F605
Neue obere Speicheradresse: F5FF
Ok
■
```

M E R G E

MERGE datei$** wird zum Anbinden der Datei **datei$ an das aktuelle Programm benutzt.

datei$ bezeichnet das anzubindende Programm. Wird die Extension weggelassen, wird **.BAS** angenommen. Auch kann vor dem eigentlichen Dateinamen eine Laufwerksbezeichnung mit angegeben werden. Während des Anbindens durch **MERGE** werden die Programmzeilen ersetzt, die mit gleicher Zeilennummer bereits existieren. Nach der Ausführung des Befehls, kehrt BASIC wieder in den Direktmodus zurück.
Zu beachten ist jedoch, daß alle Variablen, Benutzerfunktionen, **OPTION BASE-**, **DEFINT-**, **DEFSNG-**, **DEFDBL-** und **DEFSTR-**Kommandos zurückgesetzt werden, die **ON ERROR GOTO**-Funktion abgeschaltet und ein **RESTORE**-Befehl ausgeführt wird und daß alle offenen Dateien erhalten bleiben.

Beispiel:

```
10000 'Hier steht ihr Programm
MERGE "RPED.BAS
Ok
OPEN "O",#1,"M:PASS.OFF  'Dieser Teil ist nötig, da
Ok
MERGE "M:PASS.OFF        'RPED.BAS geschützt ist...
Ok
LIST
1 OPTION NOT TAB:OPTION RUN              | Dieser Teil stammt von
2 DEFINT a-z:.....                                  | RPED.BAS
     ............
10000 'Hier steht ihr Programm
Ok
■
```

M I D $

Mit **MID$ (wort$,start,länge)** läßt sich aus dem String **wort$** ein Teilstück heraussuchen, das an der Position **start** beginnt und dann eine Länge von **länge** erhält.

wort$ stellt den String dar, aus dem ein Teil herauskopiert werden soll. Er muß mindestens eine Länge von 1 Zeichen haben und kann bis zur Maximalgrenze von 255 Zeichen heranreichen.

start bezeichnet die Stelle, ab der der neue String gebildet werden soll. **start** muß eine Integerzahl zwischen 1 und 255 ergeben. Ist **start** größer als die gesamte Länge von **wort$**, wird ein neuer String mit der Länge 0 gebildet

länge gibt die Länge des herauszusuchenden Teilstrings an. Überschreitet **länge** die verbleibende Länge des Strings ab der Position **start**, reicht der neue String ab **start** bis zum Ende des ürsprünglichen Strings. Dies ist auch der Fall, wenn **länge** nicht angegeben wird (dann ist **MID$** äquivalent zu **RIGHT$**).

Beispiel:

```
10 text1$="JOYCE - mehr als ein Textsystem."
20 text2$=" PCW "
30 mitte$=MID$(text1$,9,23)
40 MID$(text1$,1,5)=text2$
50 PRINT mitte$
60 PRINT text1$
70 END
```


RUN:

```
mehr als ein Textsystem
 PCW  - mehr als ein Textsystem.
Ok
■
```

M I N

MIN (zahlenliste) ermittelt den kleinsten in **zahlenliste** enthaltenen Wert. **zahlenliste** ist in der Form **zahl1,zahl2, zahl3,..** einzugeben.

Beispiel:

```
10 a=10:b=20:c=-30
20 PRINT "MIN (";a;",";b;",";c;") ist ";MIN (a,b,c)
30 END
```

RUN:

```
MIN ( 10 , 20 ,-30 ) ist -30
Ok
■
```

M K D $ - M K U K $

Bei diesen Kommandos handelt es sich um **JETSAM**-Befehle. Genaueres erfahren Sie im **JETSAM**-Teil auf den Seiten 195 bis 197.

M O D

Mit **a MOD b** läßt sich der ganzzahlige Rest von **a** ermitteln, der entsteht, wenn man **a** durch **b** teilt.

Beispiel:

```
10 INPUT "Zahl: ",a
20 INPUT "Teiler: ",b
30 PRINT "Der ganzzahlige Rest von ";a;"durch ";b;"ist: ";a MOD b
40 END
```

RUN:

```
Zahl: 10
Teiler: 3
Der ganzzahlige Rest von  10 durch  3 ist:  1
Ok
■
```

N A M E

Durch Verwendung von **NAME altdatei$ AS neudatei$** läßt sich eine Datei umbenennen. **altdatei$** bezeichnet dabei die Datei, die umbenannt werden soll. Sie wird dann in **neudatei$** umbenannt. Wichtig ist, daß eine Datei mit dem Namen **neudatei$** nicht schon auf dem aktuellen Laufwerk existieren darf. Es ist außerdem nicht empfehlenswert, eine offene Datei umzubenennen.

Beispiel:

```
10 altname$="RPED.BAS"                  | Initialisierung
20 neuname$="R.BAS"                     |
30 DIR *.BAS                            | alle .BAS Dateien auflisten
40 PRINT                                | Neue Zeile nach DIR-Auflistung
50 NAME altname$ AS neuname$            | RPED.BAS umbenennen
60 DIR *.BAS                            | erneut alle .BAS Dateien heraussuchen
70 END                                  | Programmende
```

RUN:

```
RPED    .BAS  ... (mindestens RPED.BAS, sofern RPED.BAS auf Diskette vorhanden)
R       .BAS  ... (mindestens R.BAS, sofern RPED.BAS auf Diskette vorhanden)
Ok
■
```


N E W

Mit dem **NEW**-Kommando läßt sich ein im Speicher befindliches Programm löschen. Außerdem werden alle Variablen und Benutzerfunktionen vergessen, alle **OPTION BASE-**, **DEFINT-**, **DEFSNG-**, **DEFDBL-** und **DEFSTR-**Kommandos zurückgesetzt, die **ON ERROR GOTO**-Funktion wird abgeschaltet und ein **RESTORE**-Befehl ausgeführt. Alle offenen Dateien werden geschlossen.

Nach der Ausführung des **NEW**-Befehls springt BASIC immer in den Direktmodus zurück, da ein evtl. laufendes Programm gelöscht wurde.

Beispiel:

```
10 a=100
20 PRINT a
30 NEW
40 END
```

RUN:

```
100
Ok
LIST            | (zur Kontrolle; von Ihnen einzugeben)
Ok
PRINT a         | (zur Kontrolle; von Ihnen einzugeben)
0
Ok
■
```

N E X T

Mit **NEXT variable** wird die zu **variable** gehörige **FOR**-Schleife geschlossen (siehe **FOR**). **variable** kann auch weggelassen werden, da sich ein **NEXT**-Kommando immer auf das **letzte FOR** bezieht. Die (einzig) erlaubte Schachtelung von **FOR-NEXT** Schleifen ist:

```
FOR v1=a to b
     FOR v2=a to b
          FOR v3=a to b
               |
               |
          NEXT v3
     NEXT v2
NEXT v1
```

Beispiel:

```
10 DIM a(20,20),b(40)
20 FOR i=1 to 20
30   FOR u=20 to 1 STEP -1
40     a(i,u)=u
50   NEXT u
60 NEXT i
70 FOR i=1 to 40
80   b(i)=40-i
90 NEXT
100 PRINT "a(10,15)=";a(10,15)
110 PRINT "b(19)=";b(19)
120 END
```

RUN:

```
a(10,15)= 15
b(19)= 21
Ok
■
```

O C T $

Die Funktion **OCT$ (zahl,stellen)** dient dazu, **zahl** in einen oktalen String umzuwandeln, wobei **stellen** die Länge des oktalen Strings angibt. Falls die Länge unterschritten wird, wird der String bis zur vorgegebenen Länge mit führenden Nullen aufgefüllt. Ist der String jedoch zu lang, wird nichts abgeschnitten.

Zu beachten ist, daß **zahl** nur zwischen 0 und 65535 und **stellen** nur zwischen 0 und 16 liegen darf.

Beispiel:

```
10 a=256
20 b=65535
30 PRINT "OCT$(";a;") ergibt ";OCT$(a,4)
40 PRINT "OCT$(";b;") ergibt ";OCT$(b)
50 END
```

RUN:

```
OCT$( 256 ) ergibt 0400
OCT$( 65535 ) ergibt 177777
Ok
■
```

O N x G O S U B

Mit **ON x GOSUB zeile1,zeile2,zeile3,...** läßt sich abhängig von **x** eine der **zeilen** als Unterprogramm anspringen (siehe **GOSUB**). Beträgt **x** eins wird **zeile1** aufgerufen, bei **x**=zwei wird **zeile2** aufgerufen, bei **x**=drei **zeile3**, usw.

Beispiel:

```
10 PRINT "Bitte geben Sie eine Zahl ein (1..5):";          | zahl
20 t$=INPUT$(1):IF t$<"1" OR t$>"5" THEN 20                | wird ein
30 zahl=VAL(t$):PRINT zahl                                 | Wert zugewiesen
40 PRINT "Sie haben die Zahl ";                            |
50 ON zahl GOSUB 80,90,100,110,120                         |Sprung zum entsp. U-Prg.
60 PRINT " eingegeben."                                    |
70 END                                                     | Programmende
80 PRINT "eins";:RETURN                                    | je nach Zahl
90 PRINT "zwei";:RETURN                                    | wird der zu-
100 PRINT "drei";:RETURN                                   | gehörige String
110 PRINT "vier";:RETURN                                   | ausgegeben und
120 PRINT "fünf";:RETURN                                   | RETURN
```

RUN:

```
Bitte geben Sie eine Zahl ein (1..5):x
Sie haben die Zahl x_in_Buchstaben eingegeben.
Ok
■
```

O N x G O T O

Mit **ON x GOTO zeile1,zeile2,zeile3,...** läßt sich abhängig von **x** eine der **zeilen** anspringen (siehe **GOTO**). Beträgt **x** eins wird **zeile1** angesprungen, bei **x**=zwei wird **zeile2** angesprungen, bei **x**=drei **zeile3**, usw.

Beispiel:

```
10 PRINT "Bitte geben Sie eine Zahl ein (1..5):";          | zahl
20 t$=INPUT$(1):IF t$<"1" OR t$>"5" THEN 20                | wird ein
30 zahl=VAL(t$):PRINT zahl                                 | Wert zugewiesen
40 PRINT "Sie haben die Zahl ";                            |
50 ON zahl GOTO 60,70,80,90,100                            |Sprung zum entsp. U-Prg.
60 PRINT "eins";:GOTO 110                                  | je nach zahl
70 PRINT "zwei";:GOTO 110                                  | wird
80 PRINT "drei";:GOTO 110                                  | der entsprechende
90 PRINT "vier";:GOTO 110                                  | Wert ausgegeben
100 PRINT "fünf";:GOTO 110                                 | und zu 110 gesprungen
110 PRINT " eingegeben."                                   | Programmabschluß
120 END                                                    |
```

RUN:

```
Bitte geben Sie eine Zahl ein (1..5):x
Sie haben die Zahl x_in_Buchstaben eingegeben.
Ok
■
```

O N E R R O R G O T O

ON ERROR GOTO zeilennummer wird zur Fehlerbehandlung während eines laufenden Programms benutzt.

Prinzipiell gibt es zwei Wege, Fehler, die während eines Programms auftreten, zu behandeln: den normalen Fehlermodus (Abbruch des Programms und Ausgabe des Fehlers) und den Behandlungsmodus, der die Möglichkeit bietet, nach dem Finden eines Fehlers in ein Unterprogramm zu springen, welches mit **zeilennummer** beginnt.

Während der normale Fehlermodus von Anfang an eingestellt ist, wird der Behandlungsmodus erst durch ein **ON ERROR GOTO**-Kommando aktiviert. Ist dieser aktiviert und BASIC findet einen Fehler, werden **ERR** und **ERL** (siehe dort) gesetzt, um die Art des Fehlers zu bestimmen. Außerdem springt das Programm zu dem ab **zeilennummer** stehenden Unterprogramm, welches zweckmäßig mit dem **RESUME**-Kommando (sieh dort) abzuschließen ist. Abgeschaltet wird der Behandlungsmodus durch **ON ERROR GOTO 0**; der normale Fehlermodus ist dann wieder aktiv.

Beispiel:

```
10 ON ERROR GOTO 100                        | ab 100: Fehlerbehandlungsroutine
20 PRINT "Diese Zeile ist korrekt,"
30 PRINT "in der nächsten wird ein SYNTAX-ERROR simuliert."
40 ERROR 2
50 PRINT "Das Programm wird fortgesetzt."
60 END
100 PRINT "In Zeile";ERL;"tritt der Fehler mit der Nummer";ERR;"auf."
110 RESUME NEXT                             | Programm fortsetzen
```

RUN:

```
Diese Zeile ist korrekt,
in der nächsten wird ein SYNTAX-ERROR simuliert.
In Zeile 40 tritt der Fehler mit der Nummer 2 auf.
Das Programm wird fortgesetzt.
Ok
■
```

O P E N

Mit **OPEN art$,#dateinummer,datei$** wird eine Datei für die Ein- oder Ausgabe geöffnet und der Kennzeichnung **#dateinummer** zugewiesen.

art$** gibt an, ob die Datei zum sequentiellen Lesen oder Schreiben geöffnet werden soll. Soll sie gelesen werden, muß für **art$ "I" (I=INPUT) angegeben werden. Entsprechend muß zum Schreiben einer Datei für **art$ "O"** (O=OUTPUT) verwendet werden. Man muß jedoch darauf achten, daß beim Öffnen einer Datei zum Schreiben eine Datei mit gleichem Dateinamen überschrieben (gelöscht) wird. Ein versehentliches Verwechseln von **I** und **O** kann also fatale Folgen haben !

Nach dem Öffnungsvorgang erhält die Datei **#dateinummer** als Kennzeichnung. **dateinummer** kann jedoch nicht beliebig groß gewählt werden. Als Maximalwert voreingestellt ist 3 (maximal 3 Dateien zur gleichzeitigen Bearbeitung). Wenn man mehr gleichzeitig geöffnete Dateien braucht, kann man mit dem **CLEAR**- oder **MEMORY**-Kommando die entsprechende Zahl einstellen. Da BASIC für die Verwaltung der maximalen Dateizahl einen Teil des Speichers benutzt, sollte man die maximale Dateizahl nie überdimensionieren.

datei$** gibt an, welche Datei gelesen oder geöffnet werden soll. **datei$ muß eine eindeutige Dateikennzeichnung ergeben. Es kann jedoch auch eine Laufwerkskennzeichnung mit angegeben werden.

Beispiel:

```
10 OPEN "O",#1,"M:TEST.DAT"          | in Laufwerk M: die Datei TEST.DAT zum
                                     | Schreiben einrichten und der Dateinummer
                                     | 1 zuweisen

10 OPEN "I",#2,"PROFILE.GER"         | im aktuellen Laufwerk die Datei
                                     | PROFILE.GER zum Lesen öffnen und der
                                     | Dateinummer 2 zuweisen
```

Hinweis Das **OPEN**-Kommando wird auch zum Eröffnen von indizierten Dateien und Dateien mit wahlfreiem Zugriff benutzt. Eine genaue Beschreibung in diesem Zusammenhang befindet sich im Teil über **JETSAM** bzw. **Dateien mit wahlfreiem Zugriff** in diesem Buch.

O P T I O N B A S E

Mit **OPTION BASE anfang** läßt sich der Beginn einer Tabelle auf den durch **anfang** spezifizierten Wert festlegen, der nur 0 oder 1 betragen darf.

Wenn durch das **DIM**-Kommando eine Tabelle eingerichtet wird, beginnt sie an der durch **RUN** voreingestellten Position 0; es ist also möglich, **variable(0)** zu benutzen. In einigen Fällen, reicht es jedoch, sie erst ab der Position 1 beginnen zu lassen, nicht zuletzt um Speicherplatz einzusparen. Wichtig ist jedoch, daß nur ein **OPTION BASE**-Kommando pro Programm verwendet werden darf und daß es **vor** einem **DIM**-Kommando stehen muß.

Beispiel:

```
10 OPTION BASE 1
20 DIM a(20,20),b(40)
30 FOR i=1 to 20
40   FOR u=1 to 20
50    a(i,u)=u
60   NEXT
70 NEXT
80 FOR i=1 to 40
90   b(i)=40-i
100 NEXT
110 PRINT "a(10,15)=";a(10,15)
120 PRINT "b(19)=";b(19)
130 END
```


RUN:

```
a(10,15)= 15
b(19)= 21
Ok
■
```

O P T I O N F I E L D

Bei diesem Kommando handelt es sich um einen **JETSAM**-Befehl. Genaueres erfahren Sie im **JETSAM**-Teil auf Seite 193.

O P T I O N F I L E S

Mit **OPTION FILES einstellung$** kann das aktuelle Bezugslaufwerk und die aktuelle Usernummer geändert werden.

einstellung$** kann nur das neue Laufwerk oder die neue Usernummer enthalten. Wenn das Laufwerk geändert werden soll, muß **einstellung$ den Kennbuchstaben des neuen Laufwerks enthalten. Wenn die Usernummer geändert werden soll, muß in **einstellung$** die neue Usernummer stehen. Die vorgenommenen Änderungen bleiben jedoch nur während der Arbeit mit BASIC erhalten. Springt man wieder ins Betriebssystem zurück, sind wieder die Werte eingestellt, wie sie es vor dem Aufruf von BASIC waren.

Beispiel:

```
OPTION FILES "B"          | Laufwerk B:

OPTION FILES "5"          | Userbereich 5
```

O P T I O N I N P U T

Mit **OPTION INPUT = keyboard_status,get_character** läßt sich ein Maschinenunterprogramm für die Konsoleneingabe in BASIC einbinden.

keyboard_status bezeichnet die Adresse des Unterprogramms, welches von BASIC regelmäßig aufgerufen wird, um festzustellen, ob eine Taste gedrückt wurde (es ist daher aus Geschwindigkeitsgründen sinnvoll, es kurz zu halten). Das Ma-

schinenprogramm benötigt keine Einsprungbedingungen. Die Aussprungbedingungen sind jedoch folgende: wenn eine Taste gedrückt wurde, muß das Carry-Flag gesetzt sein (andernfalls muß es rückgesetzt sein) und das A-Register muß das Zeichen von der Tastatur enthalten. Wenn kein Zeichen anliegt, kann es jedoch auch zerstört sein. Die drei Doppelregister BC, DE und HL, sowie die restliches Flags können in jedem Fall zerstört sein. Alle anderen Register müssen jedoch immer erhalten bleiben.

get_character bezeichnet die Adresse des Unterprogramms, welches von BASIC nur dann aufgerufen wird, wenn es ein Zeichen benötigt. Die Routine muß warten, bis ein Zeichen von der Tastatur anliegt und es dann übergeben. Auch dieses Unterprogramm benötigt keine Einsprungbedingungen. Die Aussprungbedingung ist, daß das A-Register das Zeichen von der Tastatur enthält (z.B. durch die BDOS-Funktion 1). Die drei Doppelregister BC, DE und HL und alle Flags können wiederum zerstört sein, die restlichen Register müssen jedoch erhalten bleiben.

OPTION INPUT und **RUN** heben die Umleitung der Aufrufe zu Ihrer Routine auf und lassen BASIC wieder die Originalroutine benutzen.

Beispiel:

```
OPTION INPUT = &HF500,&HF5A0
```

(Auf ein ausführliches Beispiel wird an dieser Stelle verzichtet, da die Originalroutinen von BASIC diese Aufgabe ausreichend gut bewältigen. Ich nehme an, daß derjenige Leser, der eine andere Routine benötigt, in der Lage ist, sich mit diesen Informationen eine eigene Routine zu basteln.)

O P T I O N L P R I N T

Mit **OPTION LPRINT = ausgabe** läßt sich ein an den Drucker ausgegebenes Zeichen (durch **LPRINT**) auf Ihre Routine umlenken.

ausgabe bezeichnet die Adresse Ihres Maschinenunterprogramms, welches das für den Drucker bestimmte Zeichen verarbeitet. Der Einsprung: das C-Register enthält das an den Drucker zu schickende Zeichen. Der Aussprung: die Register A, BC, DE und HL sowie alle Flags können zerstört sein.

OPTION LPRINT und **RUN** heben die Umleitung der Aufrufe zu Ihrer Routine auf und lassen BASIC wieder die Originalroutine benutzen.

Beispiel:

```
10 FOR adr=&HF500 to &HF506
20   READ byte
30   POKE adr,byte
40 NEXT
50 OPTION LPRINT = &HF500
60 LPRINT "Test für den AUXOUT:-Kanal."
70 END
80 DATA &H59            :'LD E,C
90 DATA &H0E,&H04       :'LD C,04    Umleitung von LPRINT
100 DATA &HCD,&H05,&H00 :'CALL BDOS  auf den AUXOUT:-Kanal
110 DATA &HC9           :'RET
```

RUN:

```
Test für den AUXOUT:-Kanal.
```

O P T I O N N O T T A B

OPTION NOT TAB schaltet das **TAB**-Kommando (siehe dort) beim Ausdruck auf dem Bildschirm oder auf dem Drucker ab. Dieser Zustand bleibt solange erhalten, bis ein **RUN**- oder ein **OPTION TAB**-Kommando ausgeführt wird.
Die **TAB**-Abschaltung wird nötig, wenn man z.B. das Zeichen mit dem ASCII-Code 9 (↓) darstellen möchte oder verhindern will, daß bei versehentlichem Drücken der TAB-Taste während eines **INPUT**-Kommandos Verwirrung gestiftet wird, weil man nicht korrekt zurücklöschen kann (durch [←DEL]).

Beispiel:

```
10 INPUT "Probieren Sie ein paar Buchstaben und die TAB-Taste aus und drücken Sie dann
DEL",a$
20 OPTION NOT TAB
30 INPUT "Das gleiche noch einmal",a$
40 END
```

RUN:

```
Probieren Sie die TAB-Taste aus und drücken Sie dann DEL_
```
(Nach drücken der TAB-Taste können Sie nicht bis zur Ausgangsposition des Cursors zurücklöschen.)
```
Das gleiche noch einmal_
```
(Jetzt geht es.)

O P T I O N P R I N T

Mit **OPTION PRINT = ausgabe** läßt sich ein an den Bildschirm ausgegebenes Zeichen (durch **PRINT**) auf Ihre Routine umlenken.

ausgabe bezeichnet die Adresse Ihres Maschinenunterprogramms, welches das für den Bildschirm bestimmte Zeichen verarbeitet. Der Einsprung: das C-Register enthält das an den Bildschirm zu schickende Zeichen. Der Aussprung: die Register A, BC, DE und HL sowie alle Flags können zerstört sein.

OPTION PRINT und **RUN** heben die Umleitung der Aufrufe zu Ihrer Routine auf und lassen BASIC wieder die Originalroutine benutzen.

Beispiel:

```
10 FOR adr=&HF500 to &HF506
20   READ byte
30   POKE adr,byte
40 NEXT
50 OPTION PRINT = &HF500
60 PRINT "Test für den AUXOUT:-Kanal."
70 END
80 DATA &H59           :'LD E,C
90 DATA &H0E,&H04      :'LD C,04    Umleitung von PRINT
100 DATA &HCD,&H05,&H00 :'CALL BDOS  auf den AUXOUT:-Kanal
100 DATA &HC9          :'RET
```

RUN:

```
Test für den AUXOUT:-Kanal.
```

O P T I O N R U N

Durch ein **OPTION RUN**-Kommando in einem laufenden Programm wird verhindert, daß es durch ↑C (=[ALT]+[C] / [STOP]) oder ↑S (=[ALT]+[S]) angehalten werden kann. ↑C bricht normalerweise ein Programm ab, ↑S läßt es auf den nächsten Tastendruck warten). **OPTION RUN** bewirkt auch, daß das Programm schneller abläuft, da BASIC nicht mehr die Tastatur abfragen muß (nach ↑C und ↑S). Man sollte mit diesem Kommando jedoch vorsichtig umgehen. Wenn BASIC in einer Endlosschleife aktiv und **OPTION RUN** wirksam ist, bleibt einem zum Abbrechen nichts anderes übrig, als den Computer aus- und anzuschalten, was einen evtl. unwiederruflichen Datenverlust zur Folge hat.
Um **OPTION RUN** zu deaktivieren wird **OPTION STOP** verwendet.

Beispiel:

```
10 OPTION RUN
20 t$=INPUT$(1)
30 PRINT ASC(t$)
40 END
```

RUN:

```
■ (Testen Sie versch. Tasten aus, auch ↑C und ↑S !)
```

O P T I O N S T O P

OPTION STOP schaltet die Wirkung von **OPTION RUN** wieder ab, d.h. ↑C (=[ALT]+[C] / [STOP]) und ↑S (=[ALT]+[S]) haben wieder ihre alte Wirkung (↑C bricht ein Programm ab, ↑S läßt es auf den nächsten Tastendruck warten).

Beispiel:

```
10 OPTION RUN
20 'Hier steht Ihr Programm, das Sie nicht abbrechen / anhalten können
30 OPTION STOP
40 'Jetzt können Sie es wie gewohnt abbrechen / anhalten
50 END
```

O P T I O N T A B

OPTION TAB schaltet das **TAB**-Kommando (siehe dort) beim Ausdruck auf dem Bildschirm oder auf dem Drucker wieder ein, wenn es durch **OPTION NOT TAB** ausgeschaltet wurde.

Beispiel:

```
10 OPTION NOT TAB
20 INPUT "Probieren Sie ein paar Buchstaben und  die TAB-Taste aus und drücken Sie dann
DEL",a$
30 OPTION TAB
40 INPUT "Das gleiche noch einmal",a$
50 END
```

RUN:

```
Probieren Sie die TAB-Taste aus und Drücken Sie dann DEL_ (Nach Drücken der
TAB-Taste können Sie bis zur Ausgangsposition des Cursors
zurücklöschen.)
Das gleiche noch einmal_ (Jetzt geht es nicht mehr.)
```

O S E R R

Die **OSERR**-Funktion dient zur näheren Identifizierung des Fehlers 21. Da keine weiteren Informationen über diesen Fehler vorliegen und er wahrscheinlich in der CP/M-Version nie auftreten wird, hat diese Funktion keine Bedeutung.

O U T

OUT portnummer,wert sendet **wert** (0..255) zum durch **portnummer** (0..255) gewählten Ausgabekanal des Prozessors.

Beispiel:

```
10 FOR wert=0 to 255
20   OUT 246,wert
30   FOR u=0 to 30
40   NEXT
50 NEXT
60 OUT 246,0
70 END
```

RUN:

(Der aktuelle Bildschirminhalt rollt nach oben und erscheint wieder am unteren Bildschirmrand.)

P E E K

PEEK (adresse) dient zum Lesen eines Wertes aus der durch **adresse** spezifizierten Speicherstelle. **adresse** muß eine ganze Zahl zwischen 0 und 65535 (0000H..0FFFFH) ergeben. **PEEK** liefert einen Wert (Byte) zwischen 0 und 255 (00H..0FFH).

Beispiel:

```
10 PRINT CHR$(27)+"E"                    | Bildschirm löschen
20 stu$=HEX$(PEEK(&HFBF6),2)             | Stunden auslesen
30 min$=HEX$(PEEK(&HFBF7),2)             | Minuten auslesen
40 sek$=HEX$(PEEK(&HFBF8),2)             | Sekunden auslesen
50 zeit$=stu$+":"+min$+":"+sek$          | zeit$ zusammensetzen
60 PRINT CHR$(27)+"H";zeit$              | Zeit links oben ausgeben
70 IF INKEY$="" THEN 20                  | sooft wiederholen bis Tastendruck
80 END                                   | Programmende
```

RUN:

Der Bildschirm wird gelöscht, in der linken oberen Ecke erscheint die aktuelle Uhrzeit. (Der Abbruch der Uhr geschieht durch Tastendruck).
Siehe auch **POKE** (Beispiel).

P O K E

POKE adresse,wert dient zum Schreiben eines Wertes in die durch **adresse** spezifizierte Speicherstelle. **adresse** muß eine ganze Zahl zwischen 0 und 65535 (0000H.. 0FFFFH) ergeben. **wert** muß eine ganze Zahl zwischen 0 und 255 (=00H..0FFH) ergeben. Es stellt das Byte dar, welches geschrieben (ge**POKE**d) werden soll.

Beispiel:

```
10 INPUT "Stunde: ",stu$:POKE &HFBF6,VAL("&H"+stu$)
20 INPUT "Minute: ",min$:POKE &HFBF7,VAL("&H"+min$)
30 INPUT "Sekunde:",sek$:POKE &HFBF8,VAL("&H"+sek$)
40 END
```

RUN:

```
Stunde: stunden
Minute: minuten
Sekunde:sekunden
Ok
■
```

Siehe auch **PEEK** (Beispiel).

P O S (0)

POS(0) ermittelt die aktuelle logische Position des Cursors. Das heißt, daß die ermittelte Position nicht unbedingt etwas mit der tatsächlichen zu tun hat, weil jedes Zeichen, dessen ASCII-Code kleiner als 32 ist, nicht mitgezählt wird. Eine Ausnahme machen nur die Zeichen mit dem ASCII-Code 8, 9 und 13.

Die Bedeutung:

ASCII:	8	entspricht einem Rückschritt, der den Zähler um 1 verringert, falls die Position nicht schon 1 ist
	9	entspricht einem TAB, der in Leerstellen umgewandelt wird, von denen jede gezählt wird
	13	entspricht einem Wagenrücklauf, der die Position auf 1 setzt

Beispiel:

```
10 PRINT "Dies ist ein Test.";              | der Cursor befindet sich nun am Zeilenende
20 x=POS(0)
30 PRINT "Die aktuelle Position des Cursors beträgt:";x
40 END
```

RUN:

```
Dies ist ein Test.
Die aktuelle Position des Cursors beträgt 19
Ok
■
```

P R I N T

PRINT text; bewirkt die Ausgabe des **textes** auf der Konsole (meistens Bildschirm).

text kann dabei eine Variablenliste oder ein fester Ausdruck sein. Die Variablenliste kann aus Variablen vom Typ **string** und **numerisch** bestehen. Wenn die einzelnen Variablen dabei durch ein Komma getrennt werden, wird jedesmal für ein Komma ein Tabulator eingefügt. Wenn die Variablen durch ein Semikolon getrennt werden, werden die Werte der Variablen ohne zusätzliche Leerstellen aneinandergereiht gedruckt. Bei diesem Fall muß man jedoch beachten, daß eine Variable numerischen Typs eine Leerstelle vor (für Vorzeichen) und nach der Zahl beansprucht. Der feste Ausdruck ist entweder eine Zahl oder ein Text in Anführungsstrichen.

Ein Semikolon am Ende des Textes bewirkt, daß kein Wagenrücklauf ausgeführt wird. Ein Komma würde einem Tabulator gleichkommen, wobei auch hier kein Wagenrücklauf ausgeführt wird. Wenn am Ende kein Zeichen steht, wird ein ganz normaler Zeilenvorschub ausgeführt.

Beispiel:

```
10 a=8256
20 a$="SCHNEIDER JOYCE PCW"
30 PRINT "Dieses Beispiel läuft auf einem ";a$;a
40 a=10:b=20:c=-30
50 PRINT a,b,c,999;
60 PRINT "Test"
70 END
```

RUN:

```
Dieses Beispiel läuft auf einem SCHNEIDER JOYCE PCW 8256
 10            20           -30            999 Test
Ok
■
```

P R I N T

PRINT #dateinummer,text; bewirkt die Ausgabe des **textes** in eine Datei.
text kann dabei eine Variablenliste oder ein fester Ausdruck sein. Die Variablenliste kann aus Variablen vom Typ **string** und **numerisch** bestehen. Wenn die einzelnen Variablen dabei durch ein Komma getrennt werden, wird jedesmal für ein Komma ein Tabulator eingefügt. Wenn die Variablen durch ein Semikolon getrennt werden, werden die Werte der Variablen ohne zusätzliche Leerstellen aneinandergereiht gedruckt. Bei diesem Fall muß man jedoch beachten, daß eine Variable numerischen Typs eine Leerstelle vor (für Vorzeichen) und nach der Zahl beansprucht. Der feste Ausdruck ist entweder eine Zahl oder ein Text in Anführungsstrichen.

Ein Semikolon am Ende des Textes bewirkt, daß kein Wagenrücklauf ausgeführt wird. Ein Komma würde einem Tabulator gleichkommen, wobei auch hier kein Wagenrücklauf ausgeführt wird. Wenn am Ende kein Zeichen steht, wird ein ganz normaler Zeilenvorschub ausgeführt.

Beispiel:

```
10 OPEN "O",#1,"M:TEST"
20 a=8256
30 a$="SCHNEIDER JOYCE PCW"
40 PRINT #1,"Dieses Beispiel luft auf einem ";a$;a
50 a=10:b=20:c=-30
60 PRINT #1,a,b,c,999;
70 PRINT #1,"Test"
80 CLOSE
90 DISPLAY "M:TEST"
100 END
```

RUN:

```
Dieses Beispiel läuft auf einem SCHNEIDER JOYCE PCW 8256
 10            20           -30            999 Test
Ok
■
```

P U T

Bei diesem Kommando handelt es sich um einen Befehl, um in Jetsam-Dateien bzw. Dateien mit wahlfreiem Zugriff Datensätze zu schreiben. Genaueres erfahren Sie im Teil über **JETSAM** bzw. **Dateien mit wahlfreiem Zugriff** auf den Seiten 190 und 206.

R A N D O M I Z E

Mit **RANDOMIZE zahl** wird der Zufallszahlengenerator auf einen bestimmten Anfangswert gesetzt. **zahl** stellt eine beliebige ganze Zahl dar. Wird **zahl** weggelassen, reagiert BASIC mit der Frage **"Random Number Seed ?"** und fordert Sie damit zur manuellen Eingabe von **zahl** auf.

BASIC macht eine neue Zufallszahl immer von der letzten abhängig. Wenn also bei jeder Zufallszahlenreihe mit dem gleichen Anfangswert begonnen wird, sind auch die nachfolgenden "Zufallszahlen" identisch. Um diesem Manko entgegenzuwirken, gibt es das **RANDOMIZE**-Kommando, mit dem sich der Anfangswert (**zahl**) unbegrenzt variieren läßt.

Beispiel:

```
10 wert=PEEK(&HFBF8)                    | aktuelle Sekunden
20 RANDOMIZE wert
30 FOR i=1 to 20
40   PRINT INT(RND(1)*10)
50 NEXT
60 END
```

RUN:

Es erscheint eine Zufallszahlenreihe von 20 Elementen. Wenn Sie dieses Beispielprogramm mehrmals hintereinander austesten, werden Sie feststellen, daß die neue Zufallszahlenreihe in den meisten Fällen von der vorherigen abweicht Wiederholungsquote: alle 60 Läufe eine Wiederholung (da von Sekunden abhängig) .

R A N K S P E C

Bei diesem Kommando handelt es sich um einen **JETSAM**-Befehl. Genaueres erfahren Sie im **JETSAM**-Teil auf Seite 174.

R E A D

READ variablenliste liest die in den **DATA**-Zeilen gespeicherten Informationen und weist sie den in **variablenliste** genannten Variablen zu. **variablenliste** ist in der Form **variable1,variable2,variable3,...** anzugeben. Dabei ist selbstverständlich, daß die Variablentypen mit den Typen der durch READ gelesenen Informationen übereinstimmen müssen. Auch muß man darauf achten, daß man nicht versucht, hinter dem letzten **DATA**-Element zu lesen. Versucht man es trotzdem, reagiert BASIC mit dem Error 4: **DATA exhausted** (was auch sonst ?). Vgl. hierzu auch: **RESTORE**.

Beispiel:

```
10 RESTORE 130
20 FOR i=1 to 3
30  READ a
40  PRINT a
50 NEXT
60 PRINT
70 RESTORE 140
80 FOR i=1 to 4
90  READ a$
100  PRINT a$
110 NEXT
120 END
130 DATA 10,-1.234,-97531
140 DATA Test,Autobahnraststätte
150 DATA "Kommatest,Doppelpunkttest:","6,19,23,30,36,45 / 5
```

RUN:

```
 10
-1.234
-97531

Test
Autobahnraststätte
Kommatest,Doppelpunkttest:
6,19,23,30,36,45 / 5
Ok
■
```

R E M

Mit **REM kommentar** läßt sich ein BASIC-Programm mit Kommentaren versehen. Nach dem **REM**-Kommando wird jedes Zeichen als Kommentar aufgefaßt, egal ob es sich dabei tatsächlich um einen Kommentar handelt. Das **REM**-Kommando kann grundsätzlich an jeder Stelle des Programms stehen, jedoch darf es nicht hinter einem **DEL**, **ERA**, **DIR**, **REN** und **TYPE**-Kommando stehen, weil diese Befehle den Rest der Zeile als Parameter interpretieren. Statt **REM** kann auch vereinfacht ein Apostroph (') verwendet werden. Er hat die gleiche Wirkung wie **:REM**. Der Apostroph darf jedoch nicht am Ende einer **DATA**-Zeile ohne führenden Doppelpunkt stehen, da er sonst von **READ** als Beginn eines einfachen Stringes interpretiert wird.

Beispiel:

```
10 REM Dies ist ein Test

10 PRINT "Test":REM Dies ist ein Test

10 'Dies ist ein Test

10 PRINT "Test" 'Dies ist ein Test

10 DATA 1,2,3:REM Dies ist ein Test

10 DATA 1,2,3:'Dies ist ein Test
```

R E N

Durch Verwendung von **REN neuname.ext = altname.ext** läßt sich eine Datei umbenennen. **altname.ext** bezeichnet dabei die Datei, die umbenannt werden soll. Sie wird dann in **neuname .ext** umbenannt. Wichtig ist dabei, daß eine Datei mit dem Namen **neuname.ext** nicht schon auf dem aktuellen Laufwerk existieren darf. Es ist außerdem nicht empfehlenswert, eine offene Datei umzubenennen.

Beispiel:

```
10 DIR *.BAS
20 PRINT
30 REN R.BAS=RPED.BAS
40 DIR *.BAS
50 END
```

RUN:

```
RPED    .BAS  ... (mindestens RPED.BAS, sofern RPED.BAS auf Diskette vorhanden)
R       .BAS  ... (mindestens R.BAS)
Ok
■
```

R E N U M

RENUM neubeginn,altbeginn,abstand nummeriert das aktuelle Programm nach den angegebenen Parametern neu durch, wobei auch die Zeilennummern in **ELSE-**, **GOSUB-**, **GOTO-**, **LIST-**, **LLIST-**, **RESTORE-**, **RESUME-**, **RUN-**, **THEN-** und **DELETE**-Kommandos automatisch angepaßt werden. Außerdem geschieht dieser Vor-

gang bei einem **DELETE** eines **CHAIN MERGE**-Kommandos.

neubeginn bezeichnet die Zeilennummer, ab der das neu numerierte Programm abgelegt werden soll. Wird diese Angabe weggelassen, wird das Programm ab Zeile 10 abgelegt.

altbeginn bezeichnet die Zeilennummer, bei der die Neunumerierung beginnen soll. Wird diese Angabe weggelassen, beginnt die Numerierung bei der ersten Zeile des Programms.

abstand bezeichnet den Zeilenabstand, den die Zeilen nach der Neunumerierung haben sollen. Wird diese Angabe weggelassen, beträgt der Zeilenabstand 10.

Findet BASIC während der Ausführung des **RENUM**-Kommandos einen Verweis zu einer nicht vorhandenen Zeile, gibt es die Fehlermeldung **Undefined line xxxx in yyyy** aus. **xxxx** gibt die nicht existierende Zeilennummer an, **yyyy** die (neu numerierte) Zeile, in der der Fehler auftritt. Nach der Ausführung des **RENUM**-Kommandos springt BASIC in den Direktmodus zurück.

Beispiel:

```
7 '
16 '
21 '
55 '
```

(Probieren Sie die diversen Beispiele aus.)

```
RENUM          | erste Zeile 10, Zeilenabstand 10

RENUM 10       | erste Zeile 10, Zeilenabstand 10

RENUM 10,16    | erste Zeile 10 (ehemals 16), Zeilenabstand 10

RENUM ,,5      | erste Zeile 10, Zeilenabstand 5

RENUM 1,,1     | erste Zeile 1, Zeilenabstand 1
```

R E S E T

RESET laufwerk$** setzt das mit **laufwerk$ spezifizierte Laufwerk zurück und schließt dabei alle offenen Dateien.

laufwerk$ gibt das Diskettenlaufwerk an, was zurückgesetzt werden soll. Die Laufwerksbezeichnung muß dabei zwischen A und P liegen.

Beispiel:

```
RESET "A"

10 lw$="M"
20 RESET lw$
30 END
```

R E S T O R E

RESTORE zeilennummer setzt den **DATA**-Zeiger auf die durch **zeilennummer** spezifizierte **DATA**-Zeile.

zeilennummer bezeichnet die Zeile, auf die der **DATA**-Zeiger gerichtet werden soll. Die **zeilennummer** muß dabei als Konstante eingegeben werden. Wird **zeilennummer** weggelassen, wird der **DATA**-Zeiger auf das erste **DATA**-Element gesetzt. Der **DATA**-Zeiger bezieht sich auf die einzelnen **DATA**-Elemente. Mit jedem **READ**-Kommando wird er auf das folgende Element gesetzt. Das **RESTORE**-Kommando wird z.B. nötig, wenn man am Ende einer **DATA**-Liste angelangt ist und sie noch einmal (von vorne) lesen möchte.

Beispiel:

```
10 RESTORE 130
20 FOR i=1 to 3
30  READ a
40  PRINT a
50 NEXT
60 PRINT
70 RESTORE 140
80 FOR i=1 to 4
90  READ a$
100  PRINT a$
110 NEXT
120 END
130 DATA 10,-1.234,-97531
140 DATA Test,Autobahnraststätte
150 DATA "Kommatest,Doppelpunkttest:","6,19,23,30,36,45 / 5
```

RUN:

```
 10
-1.234
-97531

Test
Autobahnraststätte
Kommatest,Doppelpunkttest:
6,19,23,30,36,45 / 5
Ok
■
```

R E S U M E

Wenn ein Fehler in einem Programm auftrat und er mit einer **ON ERROR GOTO**-Routine behandelt wurde, bietet **RESUME** die Möglichkeit, das Programm normal fortzusetzen.

Dazu gibt es drei Möglichkeiten:

RESUME	**RESUME** springt zu dem Befehl zurück, bei dem der Fehler aufgetreten ist.
RESUME zeilennr	**RESUME** springt zu der durch **zeilennr** spezifizierten Zeile.
RESUME NEXT	**RESUME** springt zu dem Befehl, der unmittelbar auf den fehlerhaften folgt.

Beispiel:

```
10 ON ERROR GOTO 100   | ab 100: Fehlerbehandlungsroutine
20 PRINT "Diese Zeile ist korrekt,"
30 PRINT "in der nächsten wird ein SYNTAX-ERROR simuliert."
40 ERROR 2
50 PRINT "Das Programm wird fortgesetzt."
60 END
100 PRINT "In Zeile";ERL;"tritt der Fehler mit der Nummer";ERR;"auf."
110 RESUME NEXT                | Programm nach der fehlerhaften Zeile fortsetzen
```

RUN:

```
Diese Zeile ist korrekt,
in der nächsten wird ein SYNTAX-ERROR simuliert.
In Zeile 40 tritt der Fehler mit der Nummer 2 auf.
Das Programm wird fortgesetzt.
Ok
■
```


R E T U R N

RETURN beendet ein durch **GOSUB** oder **ON x GOSUB** aufgerufenes Unterprogramm. Nach dem **RETURN**-Kommando setzt BASIC das Programm mit dem unmittelbar auf **GOSUB** folgenden Befehl fort.

Beispiel:

```
10 PRINT "Hier läuft das Hauptprogramm."
20 PRINT "Gleich wird das Unterprogramm aufgerufen."
30 GOSUB 100
40 PRINT "Jetzt läuft das Hauptprogramm weiter."
50 END
100 PRINT CHR$(7);"---- UNTERPROGRAMM ----";CHR$(7)
110 RETURN
```

RUN

```
Hier läuft das Hauptprogramm.
Gleich wird das Unterprogramm aufgerufen.
(Pieps)---- UNTERPROGRAMM ----(Pieps)
Jetzt läuft das Hauptprogramm weiter.
Ok
■
```

R I G H T $

RIGHT$ (wort$,n) ermittelt vom rechten Ende **n** Zeichen von **wort$**. **n** muß dabei zwischen 0 und 255 liegen.
Ist **n** größer als die gesamte Länge von **wort$**, so wird **wort$** ohne Veränderungen als String übergeben.

Beispiel:

```
10 text$="JOYCE  mehr als ein Textsystem."
20 rechts$=RIGHT$(text$,24)
30 PRINT rechts$
40 END
```

RUN:

```
mehr als ein Textsystem.
Ok
■
```

R N D

Die Funktion **RND (x)** übergibt eine Zufallszahl, die nur in dem Sinne zufällig ist, als daß man die Errechnungsprinzipien für die Zufallszahl von BASIC nicht kennt. Es ist nur bekannt, daß BASIC eine Zufallszahl aus der vorherigen ableitet. Beginnt man also mit der gleichen Anfangszahl (s. **RANDOMIZE**), sind auch die nachfolgenden Zufallszahlen gleich.

Für die Form von RND gibt es vier Möglichkeiten:

RND Hier wird eine neue Zufallszahl erzeugt.

RND (positiv.x) Wie **RND (positiv.x** heißt: **positiv.x** ist größer als 0).

RND (0) Hier wird immer die letzte Zufallszahl übergeben.

RND (negativ.x) Für jedes **negativ.x** wird eine neue Zufallszahl erzeugt (**negativ.x** heißt: **negativ.x** ist kleiner als 0).

Jede Form von **RND** übergibt eine Zahl einfacher Genauigkeit, die größer gleich 0 und kleiner als 1 ist.

Beispiel:

```
10 wert=PEEK(&HFBF8) 'aktuelle Sekunden
20 RANDOMIZE wert
30 FOR i=1 to 20
40   PRINT INT(RND(1)*10)
50 NEXT
60 END
```

RUN:

Es erscheint eine Zufallszahlenreihe von 20 Elementen. Wenn Sie dieses Beispielprogramm mehrmals hintereinander austesten, werden Sie feststellen, daß die neue Zufallszahlenreihe in den meisten Fällen von der vorherigen abweicht (Wiederholungsquote: alle 60 Läufe eine Wiederholung (da von Sekunden abhängig)).

R O U N D

ROUND (zahl,stellen) rundet **zahl** auf die durch **stellen** angegebenen Vor- bzw. Nachkommastellen.

zahl stellt die Zahl dar, die gerundet werden soll. Es kann sich dabei um eine beliebige Zahl handeln.

stellen gibt an, auf wieviel Stellen vor oder nach dem Komma gerundet werden soll. Dabei ergeben sich drei Möglichkeiten:

stellen > 0 Hier wird **zahl** auf die angegebenen Nachkommastellen gerundet.

stellen = 0 **zahl** wird zu einer ganzen Zahl gerundet.

stellen < 0 **zahl** wird auf die positiven angegebenen Vorkommastellen gerundet. Dabei werden die Vorkommastellen gleich 0 gesetzt (1200000).

In jedem Fall muß **stellen** jedoch zwischen -39 und +39 liegen. Wenn Sie **stellen** nicht angeben, wird **stellen = 0** angenommen.

Beispiel:

```
10 a=1234.567
20 PRINT "ROUND (";a;", 1) ergibt: ";ROUND (a,1)
30 PRINT "ROUND (";a;", 0) ergibt: ";ROUND (a,0)
40 PRINT "ROUND (";a;",-2) ergibt: ";ROUND (a,-2)
50 END
```

RUN:

```
ROUND ( 1234.567 , 1) ergibt: 1234.6
ROUND ( 1234.567 , 0) ergibt: 1235
ROUND ( 1234.567 ,-2) ergibt: 1200
OK
■
```

R S E T

Bei diesem Kommando handelt es sich um einen **JETSAM**-Befehl. Genaueres erfahren Sie im **JETSAM**-Teil auf Seite 173.

R U N

Prinzipiell startet das **RUN**-Kommando ein vorhandenes Programm und initialisiert den Speicher, so wie er nach dem Laden von BASIC war (Löschung aller Variablen, Variableneinstellungen, etc.). Unberührt von der Initialisierung bleiben lediglich die Einstellungen des **MEMORY**-Kommandos.

RUN hat jedoch auch zwei Zusatzfunktionen:

RUN datei$** Das unter der Bezeichnung **datei$ abgespeicherte BASIC-Programm wird geladen und ausgeführt. Ist in **datei$** keine Extension angegeben, wird **.BAS** angenommen. Zusätzlich kann vor dem Dateinamen auch das Laufwerk mit angegeben werden.

RUN zeilennr BASIC beginnt die Ausführung des im Speicher befindlichen Programmes bei der angegebenen Zeilennummer (**zeilennr**) [wird **zeilennr** weggelassen, beginnt die Ausführung bei der ersten Zeilennummer].

Beispiel:

```
RUN                 | das im Speicher befindliche Programm wird gestartet

RUN "RPED"          | RPED.BAS wird geladen und gestartet

RUN "RPED.BAS"      | RPED.BAS wird geladen und gestartet

RUN 100             | das aktuelle Programm wird ab Zeile 100 gestartet
```

S A V E

SAVE "datei.ext",A sichert das im Speicher befindliche Programm unter der Dateikennzeichnung **datei.ext** auf Diskette. Ist **.ext** nicht angegeben, wird .BAS angenommen. **,A** bedeutet, daß das Programm im ASCII-Format abgespeichert wird, so daß man es z.B mit einem Texteditor bearbeiten oder mit **TYPE** ansehen könnte. Statt **,A** kann man auch **,P** verwenden, wobei das Programm in geschützter Form gespeichert wird. Nach einem späteren Laden läßt sich daß Programm nicht ohne weiteres auflisten. Will man es "entschützen", benutze man die folgende Befehlsfolge: **OPEN "O",#1,"M:PASS.OFF" :MERGE "M:PASS.OFF"**. Da diese zwei Kommandos den Schutz lahmlegen, ist seine Anwendung witzlos geworden. Benutzt man jedoch weder **,A** noch **,P**, wird das Programm im Normalformat abgespeichert, das nur von BASIC verarbeitet werden kann.

Beispiel:

```
SAVE "TEST                | das aktuelle Programm wird unter TEST.BAS gespeichert

SAVE "M:TEST              | das aktu. Programm wird in M unter TEST.BAS gespeichert

SAVE "TEST.BAS"           | das aktuelle Programm wird unter TEST.BAS gespeichert

SAVE "TEST.ASC",A    | das aktuelle Programm wird im ASCII-Format abgelegt

SAVE "TEST.PAS",P    | das aktuelle Programm wird geschützt abgelegt
```

S E E K K E Y - S E E K S E T

Bei diesen Kommandos handelt es sich um **JETSAM**-Befehle. Genaueres erfahren Sie im **JETSAM**-Teil auf den Seiten 180 bis 189.

S G N

SGN (x) [Signum (x)] prüft, ob **x** positiv, null oder negativ ist. Ist **x** positiv, übergibt **SGN** den Wert 1. Wenn **x** negativ ist, wird -1 übergeben. Wenn **x** 0 ist, wird ebenfalls 0 übergeben.

Beispiel:

```
10 a(1)=10
20 a(2)=0
30 a(3)=-6.92
40 FOR i=1 to 3
50   IF SGN(a(i))=-1 THEN PRINT a(i);"ist negativ."
60   IF SGN(a(i))=0 THEN PRINT a(i);"ist null."
70   IF SGN(a(i))=1 THEN PRINT a(i);"ist positiv."
80 NEXT
90 END
```

RUN:

```
 10 ist positiv.
 0 ist null.
-6.92 ist negativ.
Ok
■
```

S I N

SIN (x) übergibt den Sinus von **x**. **x** muß dabei größer sein als -205884.2734375 und kleiner als 205884.2734375.
Zu beachten ist, daß sich alle Angaben auf das **Bogenmaß** beziehen.

Beispiel:

```
10 x=3.1415927
20 y=4.7123889
30 PRINT "SIN (";x;") ergibt ";SIN (x)
40 PRINT "SIN (";y;") ergibt ";SIN (y)
50 END
```

RUN:

```
SIN ( 3.1415927 ) ergibt  0
SIN ( 4.7123889 ) ergibt -1
Ok
■
```

S P A C E $

SPACE$ (anzahl) erzeugt einen String mit **anzahl** Leerstellen. **anzahl** muß dabei zwischen 0 und 255 liegen.

Beispiel:

```
10 text$="JOYCE"
20 PRINT text$
30 text$=text$+SPACE$ (10)+text$
40 PRINT text$
50 END
```

RUN:

```
JOYCE
JOYCE          JOYCE
Ok
■
```


S P C

SPC (anzahl) gibt bei einem **PRINT**, **LPRINT** oder **PRINT#**-Befehl **anzahl** Leerstellen auf dem entsprechendem Medium (Bildschirm, Drucker oder Datei) aus. **SPC** kann nur in diesem Zusammenhang benutzt werden.

anzahl gibt die zu übergebenden Leerstellen an. Ist **anzahl** größer als die eingestellte Zeilenbreite des entsprechenden Mediums, wird von **anzahl** sooft die Zeilenbreite abgezogen, bis **anzahl** kleiner als die Zeilenbreite ist.

Nach einem **SPC**-Kommando wird grundsätzlich ein Semikolon angenommen, sofern es nicht schon angegeben ist. **SPC** kann natürlich aber auch von einem Komma gefolgt werden (die Wirkung von ; und , siehe **PRINT**).

Beispiel:

```
10 text$="JOYCE"
20 PRINT text$;SPC(10);text$
30 END
```

RUN:

```
JOYCE          JOYCE
Ok
■
```

S Q R

SQR (zahl) errechnet die Quadratwurzel der gegebenen **zahl**. Den Gesetzen der Mathematik folgend darf **zahl** nur größer oder gleich Null sein. Durch die rechnerischen Beschränkungen von BASIC übergibt SQR nur einen Wert einfacher Genauigkeit. **zahl** darf jedoch eine Zahl beliebigen Typs sein.

Beispiel:

```
10 zahl=256
20 PRINT "Die Quadratwurzel von";zahl;"beträgt";SQR (zahl)
30 END
```

RUN:

```
Die Quadratwurzel von 256 beträgt 16
Ok
■
```

Hinweis: **Wenn man als Ergebnis von SQR** ein "glattes" Ergebnis erwarten kann, sollte man das Ergebnis mit **ROUND** gegen evtl. Rechenungenauigkeiten von BASIC absichern (8.9999999 statt 9).

S T O P

STOP bewirkt eine Unterbrechung des laufenden Programms und läßt BASIC in den Direktmodus zurückkehren. Dieses Kommando kann dazu nützlich sein, das Programm an einer bestimmten Stelle anzuhalten, um dann z.B. Variablen zu setzen oder abzufragen. Durch Eingabe von **CONT** wird das Programm forgesetzt (siehe dort).

Beispiel:

```
10 PRINT "Dies ist ein Programm."
20 STOP
30 PRINT "Dies ist das nach STOP fortgesetzte Programm."
40 END
```

RUN:

```
Dies ist ein Programm.
Break in 20
Ok
CONT (muß von Ihnen eingegeben werden)
Dies ist das nach STOP fortgesetzte Programm.
Ok
■
```

S T R $

STR$ (zahl) wandelt **zahl** in einen String um. Er erhält das Format, wie es durch ein **PRINT**-Kommando erzeugt worden wäre (positive Zahlen erzeugen einen String mit führender Leerstelle, negative Zahlen erzeugen ein führendes Minuszeichen).

Beispiel:

```
10 a=1.234
20 b=-1.234
30 a$=STR$ (a)
40 b$=STR$ (b)
50 PRINT "*";a$;"*"
60 PRINT "*";b$;"*"
70 END
```

RUN:

```
* 1.234*
*-1.234*
Ok
■
```

S T R I N G $

STRING$ (zeichenzahl,zeichen$) oder **STRING$ (zeichenzahl, zeichen_ASCII_Code)** bildet einen String, der aus **zeichenzahl** durch **zeichen$** / **zeichen_ASCII_Code** spezifizierten Zeichen besteht.

zeichenzahl gibt an, wieviele Zeichen vom Typ **zeichen$** erzeugt werden sollen. **zeichenzahl** darf nicht größer als 255 sein, da dies die Maximallänge eines Stringes ist.

zeichen$** gibt das Zeichen an, aus welchem der neue String **zeichenzahl**-mal bestehen soll. Wenn **zeichen$ aus mehreren Zeichen besteht, wird das erste Zeichen vervielfältigt.

zeichen_ASCII_Code gibt den ASCII-Code des Zeichens an, aus welchem der neue String **zeichenzahl**-mal bestehen soll.

Beispiel:

```
10 text1$=STRING$(20,"=")
20 text2$=STRING$(20,61)                    | "=" hat ASCII-Code 61
30 PRINT text1$
40 PRINT text2$
50 END
```

RUN:

```
====================
====================

Ok
■
```

S T R I P $

STRIP$ (text$) setzt in allen Zeichen von **text$** das 7. Bit auf Null zurück. **text$** stellt einen beliebigen String dar (max. Länge: 255 Zeichen).

Erklärung: Jedes Zeichen setzt sich aus 8 Bits (0..7) zusammen (=1 Byte). Ein Bit kann nur zwei Zustände annehmen: gesetzt (=1) oder nicht gesetzt (=0). Aus der Folge der gesetzten und nicht gesetzten Bits wird eine Dezimalzahl gebildet (z.B.: 11111111 dual = 255 dezimal). Das 7. Bit ist für Zahlen zwischen 128 und 255 verantwortlich. Wird es vom gesetzten in den ungesetzten Zustand gewandelt, verringert sich die Zahl um 128: 01111111 dual = 127 dezimal. In der Praxis wird das 7. Bit z.B. für eine Dateikennzeichnung in der Extension benutzt. Ist es im ersten Zeichen der Extension gesetzt, ist die Datei mit einem Read Only-Attribut versehen. Im zweiten Zeichen kennzeichnet es das System-Attribut der Datei, im dritten das Archive-Attribut.

Beispiel:

```
10 maske$="*.*"
20 n=1
30 datei$=FIND$(maske$,n)
40 stripdat$=STRIP$(datei$)
50 IF datei$<>"" THEN PRINT "Mit Bit 7:";datei$ : PRINT "Ohne Bit 7:";stripdat$: n=n+1 :
GOTO 30
60 END
```

RUN: (evtl. LOCOSCRIPT-Disk einlegen)

```
Mit Bit 7: datei1.ext
Ohne Bit 7: datei1.ext
      ......
Ok
■
```

S W A P

SWAP (var1,var2) tauscht den Inhalt der Variablen **var1** und **var2** ohne Umweg über eine dritte Variable aus. Vorraussetzung ist, daß **var1** und **var2** vom selben Variablentyp sind.

Beispiel:

```
10 a=-4.567
20 b=123.456
30 PRINT a;b
40 SWAP a,b
50 PRINT a;b
60 END
```

RUN:

```
-4.567  123.456
123.456  -4.567
Ok
■
```

S Y S T E M

SYSTEM veranlaßt BASIC, in das Betriebssystem (CP/M Plus) zurückzuspringen. Alle offenen Dateien werden dabei geschlossen.

Beispiel:

```
10 PRINT "Ende BASIC."
20 SYSTEM
```

RUN:

```
Ende BASIC.
A>■
```

T A B

TAB (position) gibt bei einem **PRINT**, **LPRINT** oder **PRINT#** bis **position** Leerstellen auf dem entsprechendem Medium (Bildschirm, Drucker oder Datei) aus. **TAB** kann nur in diesem Zusammenhang benutzt werden.

position gibt die Druckposition an, bis zu der (von der aktuellen Position) Leerstellen ausgegeben werden. Auf dem Bildschirm z.B. entspricht **position** der Spalte, zu der der Cursor rücken soll. Ist **position** größer als die eingestellte Zeilenbreite des entsprechenden Mediums, wird von **position** sooft die Zeilenbreite abgezogen, bis **position** kleiner als die Zeilenbreite ist. Ist **position** größer als die aktuelle Druckposition, wird ein Zeilenvorschub ausgeführt und es werden Leerstellen ausgegeben, bis **position** erreicht ist.

Nach einem **TAB**-Kommando wird grundsätzlich ein Semikolon angenommen, sofern es nicht schon angegeben ist. **TAB** kann natürlich aber auch von einem Komma gefolgt werden (die Wirkung von ; und , siehe **PRINT**).

Beispiel:

```
10 text$="JOYCE"
20 PRINT text$;TAB(10);text$                    | das zweite JOYCE wird an 10. Stelle
30 END                                          | ausgegeben
```

RUN:

```
JOYCE     JOYCE
Ok
■
```

T A N

TAN (x) übergibt den Tangens von **x**. **x** muß dabei größer sein als -205884.2734375 und kleiner als 205884.2734375.

Zu beachten ist, daß sich alle Angaben auf das **Bogenmaß** beziehen.

Beispiel:

```
10 pi=4*ATN(1)
20 x=0.25*pi
30 y=0.75*pi
40 PRINT "TAN (";x;") ergibt ";TAN (x)
50 PRINT "TAN (";y;") ergibt ";TAN (y)
60 END
```

RUN:

```
TAN ( 0.7853982 ) ergibt  1
TAN ( 2.3561945 ) ergibt -1
Ok
■
```

T R O F F

TROFF (**TR**ace **OFF**) schaltet den TRACE-Modus (siehe **TRON**) ab. Das Programm läuft danach wieder normal weiter.

Beispiel:

```
10 TRON
20 PRINT CHR$(27)+"2"+chr$(0)                   | amerikanischen Zeichensatz einschalten
30 FOR i=1 to 2
40 '
50 NEXT
60 TROFF
70 PRINT CHR$(27)+"2"+chr$(2)                   | deutschen Zeichensatz einschalten
80 END
```

RUN:

```
[20] [30] [40] [50] [30] [40] [50] [60]
Ok
■
```

T R O N

TRON schaltet den TRACE-Modus ein (**TRace ON**). Er bietet die Möglichkeit, den Programmablauf Zeile für Zeile zu überwachen. Dabei wird die aktuelle Zeilennummer in eckigen Klammern (beim amerikanischen Zeichensatz) auf dem Bildschirm angezeigt, bevor die eigentliche Verarbeitung der Zeile beginnt. Abgeschaltet wird der TRACE-Modus durch **TROFF**, **CHAIN**, **LOAD**, **NEW** und **RUN datei$**.

Beispiel:

```
10 TRON
20 PRINT CHR$(27)+"2"+CHR$(0)                  | amerikanischen Zeichensatz einschalten
30 FOR i=1 to 2
40 '
50 NEXT
60 TROFF
70 PRINT CHR$(27)+"2"+CHR$(2)                  | deutschen Zeichensatz einschalten
80 END
```

RUN:

```
[20] [30] [40] [50] [30] [40] [50] [60]

Ok
■
```

T Y P E

TYPE datei.ext zeigt den Inhalt von **datei.ext** an der Konsole. **TYPE** arbeitet genau wie direkt unter CP/M Plus. Die Auflistung kann duch [ALT]+[C] (STOP) abgebrochen, durch [ALT]+[S] (f3/f4) angehalten und durch [ALT]+[Q] (f5/f6) fortgesetzt werden. Zu beachten ist, daß **datei.ext** keine Wildcards (?,*) enthalten darf, sondern eine exakte Dateispezifikation darstellen muß. Der Rest der Zeile wird von **TYPE** als Parameter interpretiert, egal ob es sich tatsächlich um solche handelt.

Beispiel:

```
10 TYPE PROFILE.GER
20 END
```


RUN:

```
(PROFILE.GER muß sich auf der Diskette im aktuellen Laufwerk befinden!)
setdef m:,* [order = (sub,com) temporary = m:]
pip
<m:=basic.com[o]
<m:=dir.com[o]
<m:=erase.com[o]
<m:=paper.com[o]
<m:=pip.com[o]
<m:=rename.com[o]
<m:=setkeys.com[o]
<m:=show.com[o]
<m:=submit.com[o]
<m:=type.com[o]
<
Ok
■
```

U N T

UNT (zahl) wandelt **zahl** in eine vorzeichenlose Integerzahl um.

zahl muß eine ganze Zahl im Bereich von 0 bis 65535 (00H - 0FFFFH) sein.

Die vorzeichenlose Integerzahl ist ein Wert im Bereich von -32768..+32767. Für **zahl**en zwischen 0 und 32767 liefert **UNT** die unveränderte Zahl. Bei 32768 und größer geht es dann mit -32767 aufwärts bis 0 weiter.

Beispiel:

```
10 INPUT "Zahl: ",a
20 IF a<0 OR a>65535 THEN 10                   | evtl. Fehleingabe abfangen
30 PRINT "UNT (";a;") = ";UNT (a)
40 END
```

RUN:

```
Zahl: x
UNT ( x ) = y
Ok
■
```

U P P E R $

UPPER$ (text$) wandelt alle Kleinbuchstaben in **text$** in Großbuchstaben um. Als Kleinbuchstaben werden alle Zeichen mit dem ASCII-Code zwischen 97 und 122 (a-z) erkannt.

Beispiel:

```
10 text$="joyce"
20 PRINT text$;" in Großbuchstaben umgewandelt ergibt ";UPPER$(text$)
30 END
```

RUN:

```
joyce in Großbuchstaben umgewandelt ergibt JOYCE
OK
■
```

U S I N G

Das **USING format$**-Kommando wird nur im Zusammenhang mit **PRINT** oder **LPRINT** benutzt. Es bewirkt die Anpassung der Ausgabe der Variablen auf dem entsprechenden Medium an das durch **format$** gekennzeichnete Format.
Für **format$** sind verschiedene Optionen einsetzbar:

Jedes # repräsentiert die Position einer Ziffer (Zahlenformat). In **format$** muß mindestens ein # enthalten sein.

. legt die Position des Dezimalpunktes fest. In **format$** darf höchstens ein . enthalten sein.

Ein , bewirkt die Aufteilung der Vorkommastellen in Dreiergruppen, die durch Komma getrennt werden. Das , muß vor den . gesetzt werden.

Mit $$ kann man vor die erste Ziffer bzw. den Dezimalpunkt ein $ setzen. Die $$ müssen vor dem Zahlenformat stehen.

Bei Verwendung von ** werden führende Leerstellen mit Sternen aufgefüllt. Die ** müssen vor dem Zahlenformat stehen.

Durch **$ werden $$ und ** kombiniert. Auch **$ muß vor dem Zahlenformat stehen.

Mit + wird das Vorzeichen der Zahl ausgegeben. Steht + vor dem Zahlenformat, wird das Vorzeichen vor die Zahl oder das $ gestellt. Steht + am Ende des Zahlenformats, wird das Vorzeichen nachgestellt.

Ein - bewirkt die nachgestellte Ausgabe eines Minuszeichens für negative oder eines Leerzeichens für positive Zahlen. - muß hinter dem Zahlenformat stehen.

Bei Verwendung von ↑↑↑↑ wird die Zahl in Exponentialdarstellung ausgegeben. ↑↑↑↑ muß direkt hinter dem Zahlenformat angegeben werden.

Beispiel:

```
10 DEFDBL a                                  | a=doppelt genau
20 a=-123456.789#                            | # wird von BASIC gesetzt
30 PRINT USING "########";a                  | im folgenden werden die verschie-
40 PRINT USING "########.######";a           | densten Formate durchgetestet
50 PRINT USING "########,.######";a          |
60 PRINT USING "$$########,.######";a        |
70 PRINT USING "$$########.######";a         |
80 PRINT USING "**########,.######";a        |
90 PRINT USING "**########.######";a         |
100 PRINT USING "**$########,.######";a      |
110 PRINT USING "**$########.######";a       |
120 PRINT USING "+$$########,.######";a      |
130 PRINT USING "$$########.######+";a       |
140 PRINT USING "$$########.######-";a       |
150 PRINT USING "$$########.########↑↑↑↑-";a |
160 END
```

RUN:

```
 -123457
 -123456.789000
 -123,456.789000
  -$123,456.789000
  -$123456.789000
***-123,456.789000
***-123456.789000
***-$123,456.789000
***-$123456.789000
   -$123456.789000
   $123456.789000-
   $123456.789000-
$123456789.0000000D-03-
Ok
■
```

V A L

VAL (zahl$) wandelt den String **zahl$** in eine äquivalente Variable numerischen Typs um.

zahl$** stellt eine Zahl dar, die durch einen String charakterisiert wird (s. **STR$). Durch **VAL** wird sie einer Variablen numerischen Typs übergeben, mit der man wieder auf übliche Weise rechnen kann.

Beispiel:

```
10 a$="12.345"
20 b$="-10.987"
30 a=VAL (a$)
40 b=VAL (b$)
50 c=a*b
60 PRINT a;b
70 PRINT c
80 END
```

RUN:

```
 12.345 -10.987
-135.6345
Ok
■
```

V A R P T R

VARPTR (variable) ermittelt die Speicheradresse von **variable** (Adresse, an der **variable** im Speicher steht).

variable ist eine Variable beliebigen Typs.

Aufbau der Variablendatenfelder im Speicher:

INTEGER: Die Variablen werden im Lowbyte- / Highbyteformat abgespeichert.

STRING: Das erste Byte bezeichnet die Länge des Strings (daher auch die maximale Stringlänge von 255 = 0FFH Zeichen). Das zweite und dritte Byte gibt die Adresse (Lowbyte- / Highbyteformat) des Strings im Speicher an.

EINFACHE GENAUIGKEIT: Die Bytes eins bis drei geben die Zahl an. Byte 4 gibt den Exponenten an. Er ist um 128 größer als der tatsächliche. Byte 4 = 0 bedeutet, daß die Zahl gleich Null ist.

DOPPELTE GENAUIGKEIT: Die Bytes eins bis sieben geben die Zahl an. Byte 8 gibt den Exponenten an. Er ist um 128 größer als der tatsächliche. Byte 8 = 0 bedeutet, daß die Zahl gleich Null ist.

Wenn statt **variable** eine Dateinummer einer geöffneten Datei angegeben ist [**VARPTR (#dateinummer)**], wird die Anfangsadresse des zugehörigen Dateipuffers übergeben.

Da sich die Variablen immer im Hauptspeicher (TPA) befinden, übergibt **VARPTR** auch nur eine Adresse zwischen 0 und 65535 (=0FFFFH).

Beispiel:

```
10 a$="Dies ist ein Test."
20 stelle=VARPTR (a$)
30 PRINT a$
40 POKE stelle,4        | vortäuschen, daß a$ nur 4 Zeichen lang ist
50 PRINT a$
60 END
```

RUN:

```
Dies ist ein Test.
Dies
Ok
■
```

V E R S I O N

VERSION (parameter) dient zur Ermittlung des verwendeten Computertyps (CP/M = JOYCE oder MS-DOS System). Da dieses Buch nur für den JOYCE geschrieben ist und hoffentlich jeder Benutzer selbst weiß, daß er einen JOYCE hat, spielt das **VERSION**-Kommando keine Rolle.