ST BASIC

Programmier-Handbuch

ATARI®

DEMENTI

Es wurden alle erdenklichen Maßnahmen getroffen, um die Richtigkeit dieser Produkt-Dokumentation zu gewährleisten. Da die Firma ATARI jedoch ständig Verbesserungen und Nacharbeiten an ihrer Computer-Hardware und -Software vornimmt, können wir keine Garantie für die Vollständigkeit und Richtigkeit dieser Dokumentation seit ihrem Erscheinen übernehmen und schließen alle Gewährleistungsansprüche aufgrund unvollständiger, unrichtiger oder nachträglich veränderter Angaben aus.

EINFÜHRUNG

BASIC ist die beliebteste und am meisten verwendete Programmiersprache. Sie ist leicht zu erlernen und dennoch ein leistungsfähiges Hilfsmittel bei der Programmierung. ST BASIC gleicht im wesentlichen den allgemeinen BASIC-Dialekten, zieht jedoch Vorteile aus der Fenstertechnik, den Drop-Down-Menüs und Grafik-Abbildern des GEM-Desktop. Diese BASIC-Version nutzt zudem die Geschwindigkeit sowie die grafischen Fähigkeiten des ST Computer-Systemes.

Das ST BASIC Programmierhandbuch ist so aufgebaut, daß der Programmierer problemlos Zugriff auf alle benötigten Informationen nehmen kann. Neulinge in der BASIC-Programmierung sollten zuerst die Beispiele in Kapitel 1 des Handbuches durcharbeiten. Hier werden die besonderen Charakteristika von ST BASIC und dem BASIC-Dektop demonstriert und erläutert.

In Kapitel 2, Anhänge, erhalten Sie leicht verständliche Erklärungen zu jedem Aspekt dieser Sprache. Dabei handelt es sich um Beschreibungen aller reservierten Wörter, der logischen Operatoren, Vorrangregeln und Fehlermeldungen. Außerdem finden Sie hier Beispielprogramme für die Mehrzahl der erklärten Begriffe.

Unabhängig davon, ob Sie gerade erst mit dem Programmieren begonnen haben, oder ob Sie bereits ein Experte auf diesem Gebiet sind, sollten Sie sich vor Ihrer Arbeit mit ST BASIC ein Sicherungsduplikat Ihrer ST BASIC-Programmdiskette anfertigen. Lesen Sie bitte im ATARI ST Bedienungshandbuch nach, um zu erfahren, wie ein Sicherungsduplikat erstellt werden kann.

INHALTSVERZEICHNIS

KAPITEL 1

EINFÜHRUNG IN ST BASIC

Das erste Kapitel dieses Handbuches beinhaltet eine allgemeine Einführung in ST BASIC und demonstriert die Arbeitsweise von ST BASIC innerhalb der Desktop-Oberfläche des ST Computer-Systemes.

Das Kapitel ist in drei Abschnitte untergliedert:

- Ladeanweisungen
- Überblick über das GEM-Desktop
- Schreiben von Programmen in ST BASIC

Anmerkung: Bevor Sie damit beginnen, in ST BASIC zu programmieren, sollten Sie sich ein Sicherungsduplikat Ihrer Programmdiskette anfertigen. Damit schützen Sie sich vor einem Verlust der Daten auf Ihrer ST BASIC-Diskette, sofern die Programmdiskette versehentlich einmal gelöscht oder zerstört wird. Lesen Sie bitte im ATARI ST Bedienungshandbuch nach, um zu erfahren, wie ein Sicherungsduplikat angefertigt werden kann.

LADEANWEISUNGEN

Bevor sie mit ST BASIC arbeiten können, müssen Sie die Programmdiskette in den Arbeitsspeicher des ST Computers laden. Befolgen Sie hierzu die nachstehenden Anweisungen. Wenn Sie ein Computer-System mit nur einem Laufwerk besitzen, lesen Sie bitte die Instruktionen unter "Ladeanweisung bei einem angeschlossenen Diskettenlaufwerk". Demzufolge gelten die Hinweise unter "Ladeanweisung bei zwei angeschlossenen Laufwerken" für Computer-Systeme mit zwei Diskettenlaufwerken.

Ladeanweisung bei einem angeschlossenen Laufwerk

1. Schalten Sie den ST Computer ein. Wenn das GEM-Desktop auf dem Bildschirm zu sehen ist, klicken Sie zweimal auf das "Diskstation B"-Abbild.

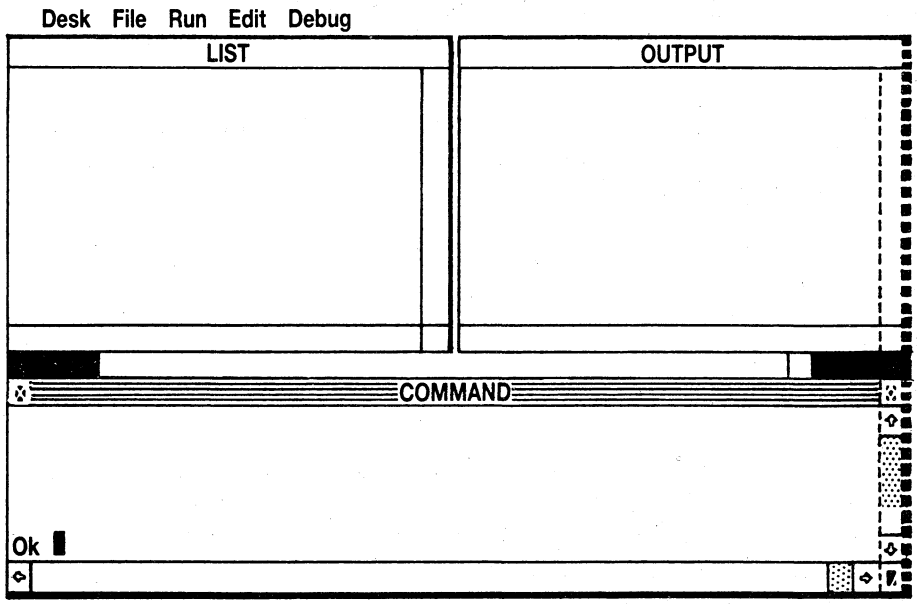
2. Sobald Sie über ein Dialogfeld dazu aufgefordert werden, die Diskette B in Laufwerk A einzulegen, entfernen Sie die TOS Systemdiskette aus Laufwerk A und legen die ST BASIC-Programmdiskette ein.
   Drücken Sie dann die RETURN-Taste.

3. Nachdem sich das Diskettenfenster geöffnet hat, klicken Sie zweimal auf das "BASIC.PRG"-Abbild. Daraufhin erscheint das BASIC-Desktop auf dem Bildschirm.

Ladeanweisung bei zwei angeschlossenen Laufwerken

1. Schalten Sie den ST Computer ein. Wenn das GEM-Desktop auf dem Bildschirm zu sehen ist, legen Sie die ST BASIC-Programmdiskette in Laufwerk B ein und klicken zweimal auf das "Diskstation B"-Abbild.

2. Nachdem sich das Diskettenfenster geöffnet hat, klicken Sie zweimal auf das "BASIC.PRG"-Abbild. Daraufhin erscheint das BASIC-Desktop auf dem Bildschirm.



Das BASIC-Desktop ist Ihr wichtigster Bezugspunkt für alle Arbeiten mit ST BASIC. In den nächsten beiden Abschnitten dieses Kapitels wird beschrieben, wie ein kleines Programm in ST BASIC geschrieben wird, und wie das BASIC-Desktop für die Arbeit mit ST BASIC verwendet werden kann.

ÜBERBLICK ÜBER DAS GEM-DESKTOP

ST BASIC arbeitet mit den Standard-Operationen des GEM-Desktop. Die Arbeitsschritte für den Zugriff auf Menübegriffe, die Auswahl von Optionen, die Handhabung von Fenstern und die Ladevorgänge werden detailliert im ATARI ST Bedienungshandbuch beschrieben.

Fenster

Für die Arbeit mit ST BASIC stehen Ihnen vier unterschiedliche Fenster zur Verfügung: Das Befehlsfenster (Command), das Ausgabefenster (Output), das Auflistungsfenster (List) und das Bearbeitungsfenster (Editor). Nachdem Sie das Programm ST BASIC geladen haben und das BASIC-Desktop auf dem Bildschirm zu sehen ist, ist das Befehlsfenster aktiviert. Alle vier Fenster sind verfügbar (wobei das Bearbeitungsfenster zu einem Großteil von den anderen Fenstern überlagert wird und daher nur teilweise zu sehen ist).

Die Verfahren für das Bewegen, Vergrößern, Öffnen, Schließen, Rollen und Anordnen von Fenstern entsprechen den Methoden, die in Kapitel 4 des ATARI ST Bedienungshandbuches beschrieben sind. Bitte lesen Sie dort nach, um weitere Informationen zu diesem Thema zu erhalten.

Das Befehlsfenster
BASIC-Befehle und Programmzeilen werden in das Befehlsfenster eingegeben. Die Anfrage "Ok" zeigt an, daß ST BASIC bereit für eine Befehlseingabe ist. Geben Sie

PRINT "HALLO"

ein und drücken Sie RETURN. Das Wort "HALLO" erscheint nun im Ausgabefenster. Geben Sie Ihren Namen ein und drücken Sie die RETURN-Taste, um die Arbeitsweise von ST BASIC kennenzulernen.

Anmerkung: Wenn Sie eine Eingabe vornehmen, die ST BASIC nicht kennt, erscheint die Fehlermeldung "Something is wrong" (fehlerhafte Eingabe) im Befehlsfenster. Durch ein Exponentialzeichen (^) wird die Stelle innerhalb der Programm-Anweisung gekennzeichnet, bei der ST BASIC den Fehler lokalisiert hat. Eine vollständige Auflistung aller vorkommenden Fehlermeldungen von ST BASIC finden Sie in Anhang D.

Ihr ST Computer kann auch die Funktion eines Taschenrechners übernehmen. Geben Sie beispielsweise

PRINT 2+2 [RETURN]

in das Befehlsfenster ein. Im Ausgabefenster erscheint das Ergebnis, 4.

Sie können für Ihre Berechnungen auch den numerischen Tastaturblock verwenden. Geben Sie ein:

? [Leertaste]

Geben Sie dann über den numerischen Tastaturblock

(5+3)*(6+2)/4+2 [Enter]

ein. Das Ergebnis, 18, erscheint wiederum im Ausgabefenster. Beachten Sie hierbei, wie ST BASIC arithmetische Operationen handhabt. Die Reihenfolge der einzelnen Rechenarten ist

1. Multiplikation
2. Division
3. Addition
4. Subtraktion

Anmerkung: Die Schreibweise eines Wortes in eckigen Klammern innerhalb eines Programm-Beispieles (z.B. [RETURN] oder [Esc] bedeutet, daß Sie die angegebene Taste auf der ST Tastatur betätigen sollen.

Das Ausgabefenster
Das Ausgabefenster von ST BASIC wird dazu verwendet, um die Resultate eingegebener Befehle oder von Programm-Operationen anzuzeigen. Alle Programmeingaben und Ausgaben an den Monitor erscheinen in diesem Fenster.

Geben Sie

INPUT A

ein. Sobald Sie die RETURN-Taste drücken, erscheint ein Fragezeichen im Ausgabefenster. Wenn Sie nun die Zahl 2 eintippen, erscheint diese im Ausgabefenster. Drükken Sie dann RETURN. Im Befehlsfenster ist wieder die Anfrage "Ok" zu sehen.

Geben Sie

10 PRINT "HALLO" [RETURN]

ein. Sie haben gerade ein einzeiliges BASIC-Programm geschrieben. Geben Sie

RUN [RETURN]

ein. Das Wort "HALLO" erscheint im Ausgabefenster.

Das Auflistungsfenster
Geben Sie

LIST [RETURN]

ein. Ihr einzeiliges Programm erscheint daraufhin im Auflistungsfenster. In diesem Fenster wird immer das Programm angezeigt, das sich derzeit im Arbeitsspeicher des Computers befindet. Wenn Sie einen Drucker an Ihren ST Computer angeschlossen haben, können Sie durch Eingabe von LLIST eine Auflistung Ihres Programmes über den Drucker ausdrucken lassen.

Das Bearbeitungsfenster
Geben Sie

EDIT [RETURN]

ein. Ihr Programm wird nun im Bearbeitungsfenster dargestellt. Veränderungen am Programm können nur innerhalb dieses Fensters vorgenommen werden. Lesen Sie bitte unter dem Abschnitt "Schreiben eines Programmes in ST BASIC" nach, um ausführlichere Informationen über das Bearbeitungsfenster zu erhalten. Durch Betätigen der Funktionstaste [F10] verlassen Sie den Editor.

Menüs

Die Menüleiste erstreckt sich über den oberen Rand des ST Desktop. Die Menütitel lauten "Desk", "File", "Run", "Edit" und "Debug". Für jeden Menütitel existiert ein eigenes Menü. Um die einzelnen Menüoptionen ablesen zu können, richten Sie den Maus-Zeiger auf den Menütitel. Das Menü wird automatisch unter dem Menütitel hervorgezogen. Wollen Sie keinen Menübegriff auswählen, klicken Sie auf eine freie Stelle des ST BASIC-Desktop. Daraufhin verschwindet das Menü wieder unter seinem Menütitel.

Dialogfelder und Fehlermeldungen

Dialogfelder erscheinen auf dem ST BASIC-Desktop, sobald das Programm Informationen von Ihnen benötigt, die aus dem Programmlisting nicht zu entnehmen sind. Falls eine Fehlermeldung dargestellt wird, wird eine Information angezeigt, die sich auf ein ST BASIC-Format oder -Programm bezieht. Eine vollständige Auflistung aller vorkommenden Fehlermeldungen von ST BASIC finden Sie in Anhang D.

Um ein Dialogfeld verlassen zu können, zeigen Sie auf eines der beiden "Exit"-Felder und klicken einmal die linke Maustaste. Ist ein "Exit"-Feld mit einem verstärkten Rand versehen, entspricht ein Betätigen der RETURN-Taste einem Klicken in dieses Feld.

Sonderfunktionen

In ST BASIC stehen Ihnen drei Sonderfunktionen zur Verfügung, durch die das Eingeben und Lesen Ihrer Programme vereinfacht werden kann: die Funktionen AUTO und RENUM, sowie die Verwendung von Sprungmarken.

AUTO Zeilennummer
Geben Sie

AUTO [RETURN]

ein. Im Befehlsfenster erscheinen zwei Sternchen und die Zahl 10. Die Zahl 10 ist die erste Zeilennummer, die von der AUTO-Funktion generiert wurde. Die beiden Sternchen signalisieren, daß im Arbeitsspeicher bereits eine Programmzeile mit der Nummer 10 existiert.

Drücken Sie die RETURN-Taste. ST BASIC wartet nun auf die Eingabe der Programmzeile 20. Da Sie bisher noch keine Zeile mit der Nummer 20 eingegeben hatten, befinden sich vor der Zahl keine Sternchen.

Geben Sie

PRINT "ICH BIN EIN TOLLER ATARI COMPUTER" [RETURN]

ein. Im Arbeitsspeicher befindet sich jetzt ein zweizeiliges Programm. Um die automatische Zeilennumerierung abzuschalten, betätigen Sie die Tastenkombination [CONTROL][G].

Im Befehlsfenster erscheint wieder die Anfrage "Ok". Geben Sie LIST ein, um Ihr Programm aufzulisten. Da Zeile 20 für die Darstellung im Auflistungsfenster zu lang ist, müssen Sie in das Größeneinstellungsfeld in der unteren rechten Ecke des Auflistungsfensters klicken und das Fenster so weit vergrößern, bis das gesamte Programmlisting sichtbar ist.

Renum
ST BASIC verfügt über eine RENUM-Funktion, über die Sie Ihr Programm automatisch neu numerieren lassen können. RENUM nimmt Zugriff auf die Diskette. Aus diesem Grund sollten Sie vor Verwendung dieser Funktion sicherstellen, daß sich eine Diskette im Laufwerk befindet.

Anmerkung: Die Funktion RENUM arbeitet nicht, wenn die eingelegte Diskette mit einem Schreibschutz versehen ist. Weitere Informationen hierzu erhalten Sie in Kapitel 6 des ATARI ST Bedienungshandbuches.

Geben Sie

RENUM 30,10,5 [RETURN]

ein. Sobald die Anfrage "Ok" auf dem Bildschirm erscheint, können Sie Ihr Programm durch Eingeben von LIST auflisten lassen. Die ehemalige Zeile 10 hat jetzt die Nummer 30. Die Erhöhungen der Zeilennummern erfolgen in Fünferschritten. Deshalb wird die nächste Zeilennummer 35 sein. Nähere Erklärungen zur RENUM-Funktion erhalten Sie in Anhang C.

Sprungmarken
ST BASIC gestattet die Verwendung von Sprungmarken (Labels) für die Kennzeichnung von Programmzeilen. Eine Anweisung GOTO DONE ist beispielsweise leichter zu lesen als GOTO 300 und erleichtert ein Nachvollziehen der Auswirkungen bestimmter Programmzeilen auf Ihren Programmlauf.

SCHREIBEN VON PROGRAMMEN IN ST BASIC

In diesem Abschnitt erfahren Sie, wie einfache Programmiertechniken innerhalb der GEM-Benutzeroberfläche eingesetzt werden. Befolgen Sie die nachstehenden Anweisungen sorgfältig.

Anmerkung: Sie können ST BASIC-Programme in Großbuchstaben oder in normaler Schreibschrift eingeben.

Eingabe
Sofern sich im Auflistungsfenster bereits ein Programmlisting befindet, löschen Sie dieses Listing durch Eingeben von

CLEARW 1

Geben Sie dann

NEW [RETURN]

ein. Dadurch wird ein im Arbeitsspeicher des Computers befindliches Programm gelöscht. Geben Sie

LIST [RETURN]

ein. Das Auflistungsfenster müßte nun unbeschrieben sein. Geben Sie

AUTO [RETURN]

ein und schreiben Sie das nachstehende Programm. Beachten Sie, daß die Zeilennummern von ST BASIC vorgegeben werden. Sie müssen diese Zahlen nicht selbst eingeben.

```
10 REM ZAEHL.BAS
20 C=0
30 ZAEHL:' ERHOEHUNG DER VARIABLEN C
40 C=C+1
50 PRINT C;
60 IF C=5 THEN PRINT "WIEDERHOLUNG!":GOTO 20
70 GOTO ZAEHL
```

Das Programm ZAEHL.BAS befindet sich nun im Arbeitsspeicher.

Geben Sie [CONTROL][G] ein, um AUTO abzuschalten.

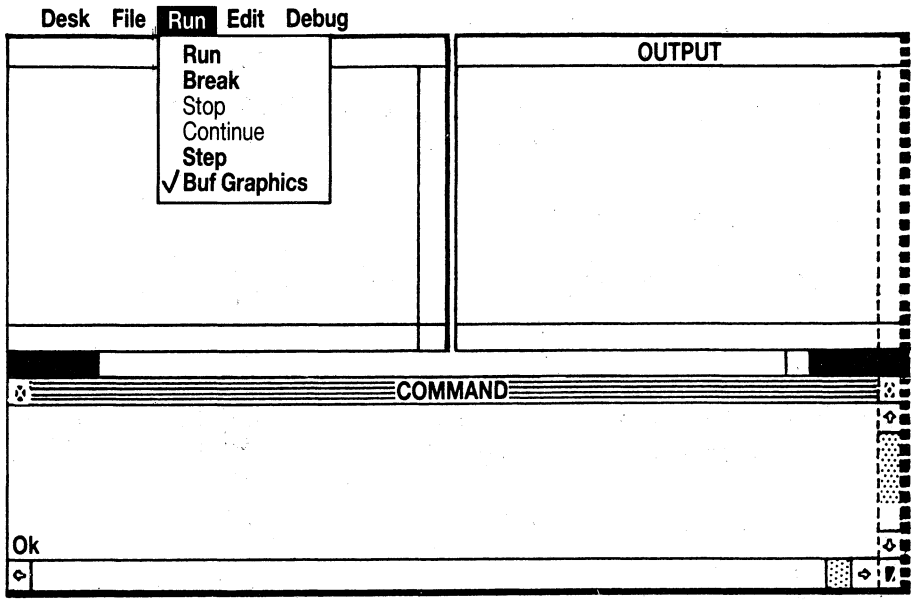
Anhand dieses einfachen Programmes werden bereits einige Funktionen von ST BASIC illustriert.

In Zeile 10 steht eine Anmerkung (REM), die die nachfolgende Funktion verdeutlichen soll. Das REM wird von ST BASIC nicht beachtet. Sie können anstelle von REM auch ein einfaches Anführungszeichen (') verwenden (siehe Zeile 30).

Zeile 30 wird durch die Sprungmarke ZAEHL identifiziert. In Zeile 70 wird dieselbe Sprungmarke innerhalb einer GOTO-Anweisung verwendet. Bei der ersten Definition muß eine Sprungmarke von einem Doppelpunkt (:) gefolgt werden. Eine Sprungmarke darf kein reserviertes Wort von ST BASIC sein, muß mit einem Buchstabenzeichen beginnen und darf keine Leerstellen enthalten.

In Zeile 60 wird gezeigt, wie das Doppelpunktzeichen eingesetzt werden kann, um mehrere Befehle in eine Programmzeile schreiben zu können. Sie können beliebig viele Befehle in eine Programmzeile setzen, solange diese durch Doppelpunktzeichen voneinander abgetrennt werden und die Zeile nicht länger als 249 Zeichen wird.

Programmlauf



Öffnen Sie das Menü "Run" und klicken Sie auf "Run". Daraufhin wird im Ausgabefenster wiederholt

1 2 3 4 5 WIEDERHOLUNG!

ausgegeben. Um das Programm anzuhalten, klicken Sie auf "Break" im Menü "Run". Durch die Mitteilung --Break -- at line .. erfahren Sie, in welcher Programmzeile der Programmlauf abgebrochen wurde. Geben Sie STOP [RETURN] ein, um den Break-Modus zu verlassen. Innerhalb des Break-Modus können sämtliche Programmier-Befehle verwendet werden.
Sie können Ihr Programm zeilenweise ausführen lassen. Dazu wählen Sie "Step" aus dem Menü "Run" aus. Nach jedem Betätigen der RETURN-Taste wird die nächste Programmzeile ausgeführt. Beachten Sie, daß die Nummer der gerade ausgeführten Programmzeile im Befehlsfenster dargestellt wird. Geben Sie nun

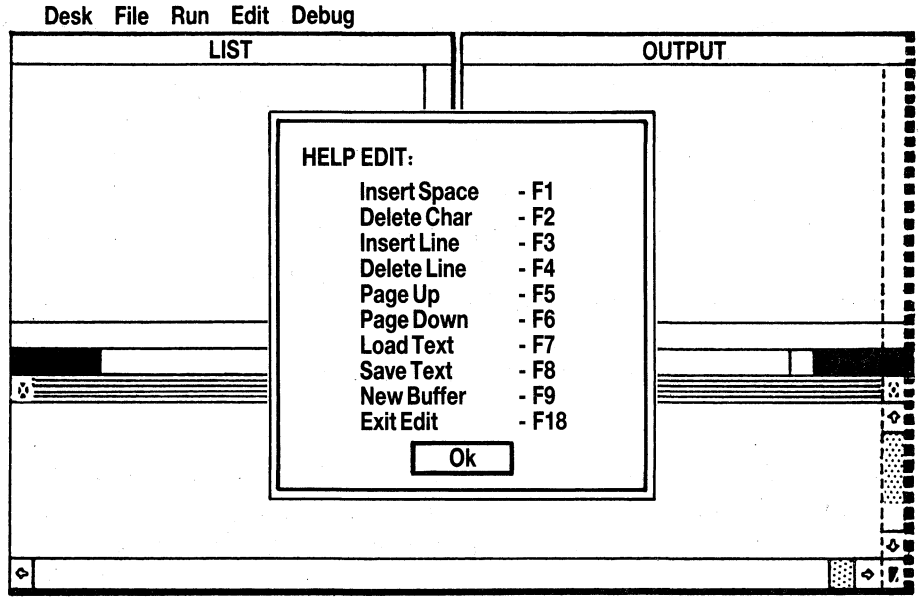
END [RETURN]

ein, um die Option "Step" abzuschalten.

Editieren von Programmen
ST BASIC verfügt über einen leicht zu bedienenden Editor, mit dem Veränderungen an Ihren Programmen vorgenommen werden können, ohne dabei eine vollständige Programmzeile neu eingeben zu müssen. Um ein Programm zu editieren, wählen Sie das Menü "Edit" aus und klicken auf die Option "Start Edit". (Sie können auch ED eingeben.)

Bringen Sie den Cursor zum Buchstaben "W" von "Wiederholung" in Zeile 60. Jetzt können sie dieses Wort überschreiben. Schreiben Sie stattdessen "Weiter". Beachten Sie, daß das Schriftbild sich verändert,um kennzuzeichnen, daß Sie in dieser Programmzeile Änderungen vorgenommen haben, die noch nicht in den Programmspeicher übernommen wurden. Drücken Sie die RETURN-Taste. Sie müssen nun noch die verbliebenen Buchstaben "holung" löschen.

Die Funktionstasten
Bevor Sie mit Ihrer Arbeit fortfahren, sollten Sie die Option "Help Edit" im Menü "Edit" auswählen.



Aus dem daraufhin erscheinenden Dialogfeld können Sie eine Beschreibung aller Funktionstasten-Befehle von ST BASIC entnehmen.

Klicken Sie in das "Ok"-Feld, um das Dialogfeld auszublenden.

Im nachfolgenden Beispiel werden die Funktionstasten für die Editierung des Programmes verwendet. Sie können jedoch auch mit der Maus und den Optionen des Menüs "Edit" arbeiten, wenn Sie dies vorziehen.

Zeichen löschen/einfügen (Delete Char/Insert Space)
Während sich der Cursor über dem ersten Buchstaben des zu löschenden Wortteiles "holung" befindet, drücken Sie die Funktionstaste [F2]. Jedes Drücken von [F2] löscht das derzeit unter dem Cursor befindliche Zeichen und verschiebt die rechts daneben-liegenden Zeichen um eine Position nach links. Löschen Sie nun alle Zeichen auf diese Weise.

Bewegen Sie den Cursor auf den Buchstaben "W" von "Weiter". Drücken Sie fünfmal die Funktionstaste [F1]. Geben Sie dann

MACH

ein. In der Programmzeile steht nun:

60 IF C=5 THEN PRINT "MACH WEITER!":GOTO 20

Neuer Speicherinhalt (New Buffer)
Wenn Sie [RETURN] drücken, werden die im Bearbeitungsfenster dargestellten Programmzeilen in den Programmspeicher übertragen. Um zu sehen, was sich derzeit im Programmspeicher befindet, drücken Sie [F9], New Buffer. Der Inhalt des Programmspeichers wird daraufhin in das Bearbeitungsfenster einkopiert. Haben Sie die RETURN-Taste noch nicht gedrückt, erscheint im Editierfenster Ihr ursprüngliches Programm ohne die gerade vorgenommenen Veränderungen.

Zeile einfügen/löschen (Insert Line/Delete Line)
Bewegen Sie den Cursor in Zeile 30. Drücken Sie dann [F4]. Zeile 30 wird daraufhin aus dem Programmspeicher entfernt, wie Sie anhand des veränderten Schriftbildes erkennen können. Diese Zeile verbleibt allerdings solange im Bearbeitungsfenster, bis Sie [RETURN] drücken. Diese Funktion vereinfacht Korrekturarbeiten erheblich. Bringen Sie den Cursor einfach in Zeile 30 und drücken Sie [RETURN]. Sobald Sie [F9] für "New Buffer" drücken, wird die Zeile sowohl im Programmspeicher, als auch im Editier-Speicher gelöscht.

Drücken Sie [F9] für "New Buffer". Zeile 30 wird nun gelöscht.

Bewegen Sie den Cursor in Zeile 50 und drücken Sie [F3] für "Insert Line". Dadurch wird Platz für die Eingabe einer neuen Programmzeile geschaffen.

Da die Numerierung Ihrer Programmzeilen langsam unordentlich wird, sollten Sie sie neu durchnumerieren lassen.

Schaffen Sie zuerst durch Drücken von [F3] Platz für eine neue Programmzeile. Geben Sie dann RENUM [RETURN] ein. Sobald der Cursor wieder sichtbar wird, können Sie "New Buffer" aufrufen, um zu sehen, wie Ihr Programm nun numeriert ist.

Durch Ihre Veränderungen hat sich ein Fehler eingeschlichen. In Zeile 70 steht "GOTO ZAEHL", aber die Zeile mit der Sprungmarke "ZAEHL" wurde von Ihnen soeben gelöscht.

Verändern Sie Zeile 30 wie folgt:

```
30 ZAEHL:C=C+1
```

Sie können den Programmlauf direkt über das Bearbeitungsfenster aufnehmen lassen. Schaffen Sie Platz für eine neue Zeile und geben Sie

RUN [RETURN]

ein.

Mit [CONTROL][C] kann das Programm angehalten werden. Sie kehren damit wieder ins Bearbeitungsfenster zurück.

Text laden/speichern (Load Text/Save Text)
Der ST BASIC Editor speichert den Inhalt des Bearbeitungsfensters auf Diskette. Allerdings ist die Speicherkapazität hier auf 24 Textzeilen beschränkt. Umfaßt Ihr Programm mehr als 24 Zeilen, werden die außerhalb des Fensters liegenden Textzeilen nicht auf Diskette abgelegt.

Anmerkung: Diese Funktion unterscheidet sich von der Funktion "Save As" im Menü "File". Mit Hilfe der Funktion "Save As" können Sie vollständige Programme speichern, die dann wieder geladen und gestartet werden können. Bei Verwendung der Funktion "Load Text/Save Text" kann zudem kein Dateiname angegeben werden. Der gespeicherte Text muß nicht unbedingt ein ST BASIC-Programm sein.

Betätigen Sie die Funktionstaste [F8] für "Load Text". Dann drücken Sie [RETURN] für jede Programmzeile. Wenn Sie jetzt [F9] drücken, befindet sich Ihr Programm sowohl im Bearbeitungsfenster, als auch im Programmspeicher.

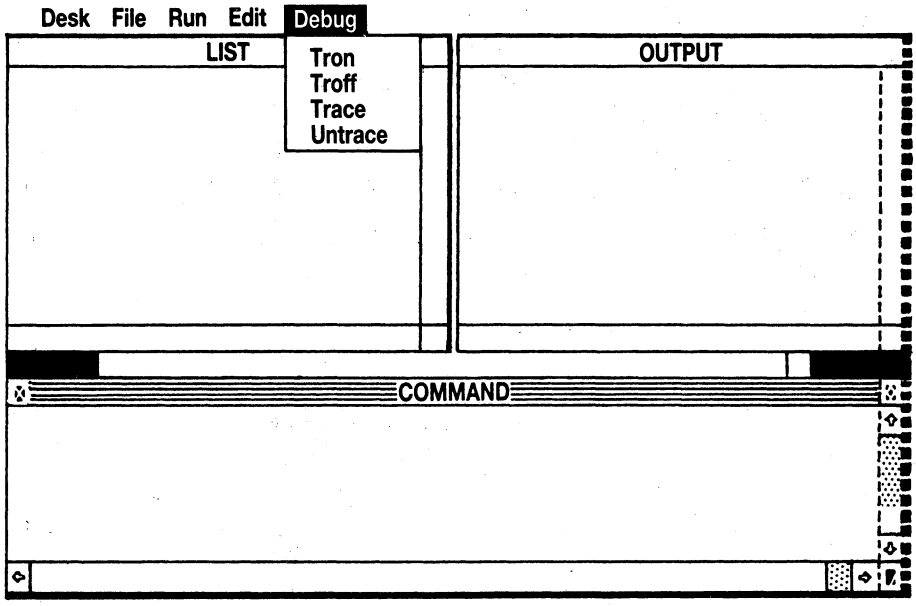
Vorhergehende/Nachfolgende Seite (Page Up/Page Down)
Die Funktionen Page Up [F5] und Page Down [F6] ermöglichen die Bearbeitung von Programmen, die mehr als einen Fensterausschnitt umfassen. Mit Page Up [F5] werden Programmzeilen sichtbar gemacht, die sich überhalb des aktuellen Fensterausschnittes befunden hatten. Umgekehrt können Sie mit [F6] in die letzten beiden Programmzeilen gelangen.

Anmerkung: Die maximal sichtbare Zeilenlänge umfaßt 80 Zeichen. Falls Sie über den rechten Rand des sichtbaren Fensterausschnittes hinausschreiben, verschiebt sich der Text im Fenster nach links, um diese Eingaben sichtbar zu machen. Im Bearbeitungsfenster können Sie maximal 80 Zeichen pro Zeile eingeben. Wenn Sie beabsichtigen, ein Programm zu editieren, das Zeilen mit mehr als 80 Zeichen enthält, wird der Teil der Zeile, der hinter dem achtzigsten Zeichen liegt, in die darunterliegende Zeile geschoben. Dabei wird dieser Teil nur dann als Teil der darüberliegenden Zeile angesehen, wenn das erste Zeichen der zweiten Zeile ein Leerzeichen ist. Andernfalls müssen Sie die Zeilensegmente so verändern, daß Sie sie als zwei separat numerierte Programmzeilen eingeben können.

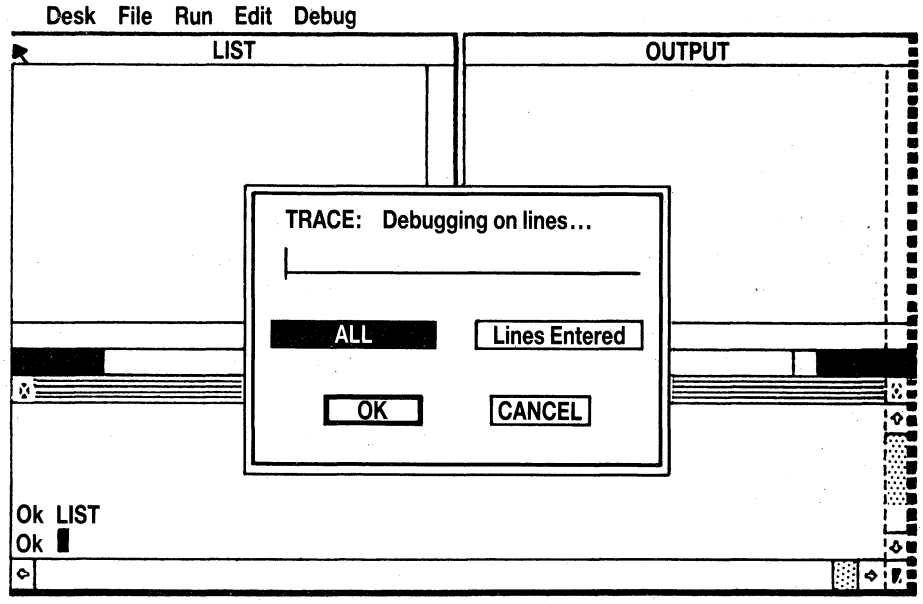
Sie können den Editor durch Klicken auf die Funktion "Exit Edit", oder durch Drükken von [F10] verlassen.

Fehlerbehebung
Mit den Möglichkeiten, die Ihnen innerhalb des Menüs "Debug" zur Verfügung stehen, ist die Beseitigung von Fehlern eine problemlose Tätigkeit. Zwei Optionen des Menüs "Debug" helfen Ihnen dabei, festzustellen, was ein Programm gerade tut und welches Problem aufgetreten sein könnte. Bei diesen Optionen handelt es sich um "Trace" und "Tron".

Wählen Sie das Menü "Debug" aus.



Klicken Sie auf die Option “Trace“.

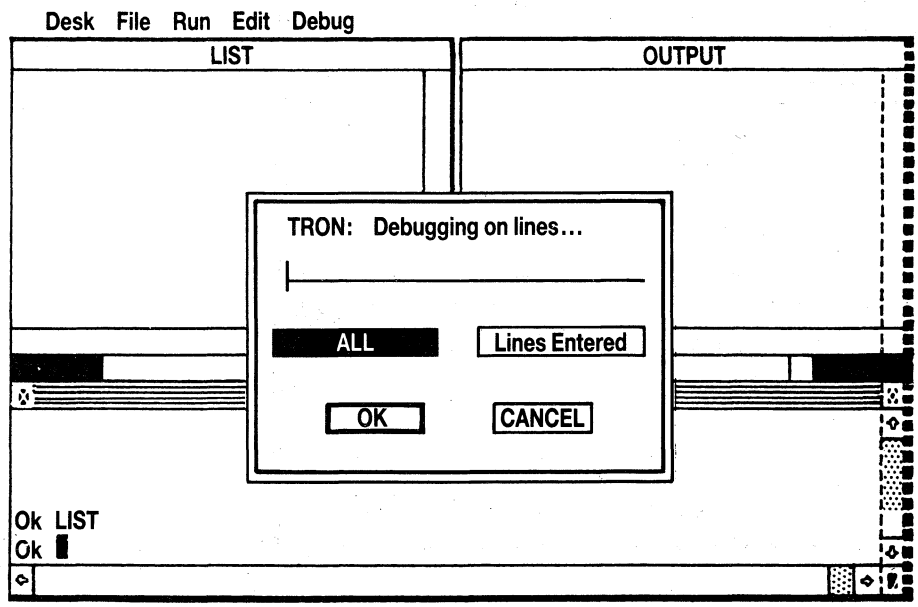


Klicken Sie in das “Ok“-Feld im Dialogfeld.

Jetzt können Sie Ihr Programm mit “Run“ ablaufen lassen. Während eine Programmzeile ausgeführt wird, listet “Trace“ diese Zeile im Befehlsfenster auf.

Um die Option “Trace“ zu verlassen, halten Sie das Programm an, öffnen das Menü “Debug“ und wählen “Untrace“ aus. Bestätigen Sie Ihre Wahl durch Klicken in das “Ok“-Feld im “Trace“-Dialogfeld.

Klicken Sie auf die Option “Tron“ im Menü “Debug“.



"Tron" zeigt die Nummer der derzeit in Ausführung befindlichen Programmzeile an.

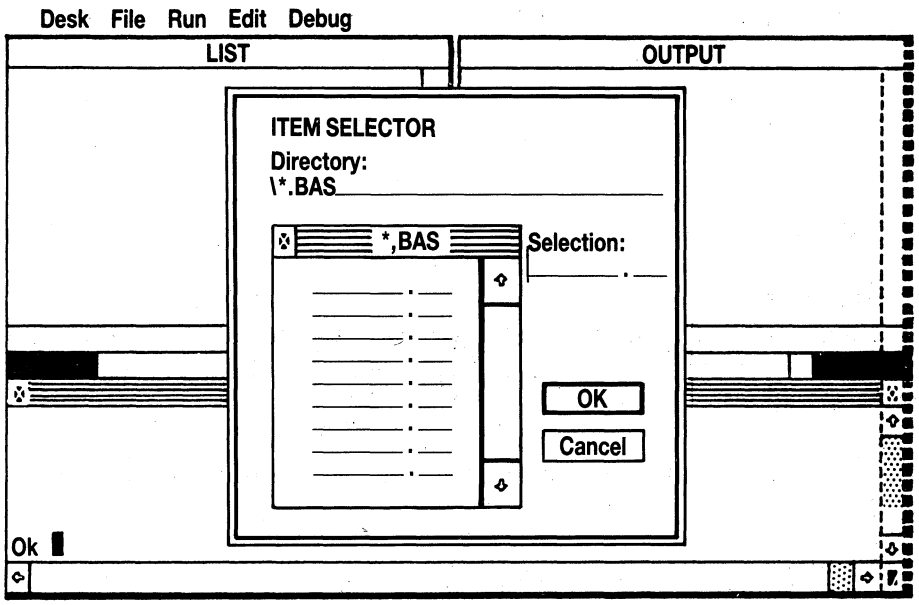
Klicken Sie in das "Ok"-Feld im Dialogfeld.

Starten Sie Ihr Programm nochmals. Während die Programmzeilen nacheinander ausgeführt werden, erscheinen die entsprechenden Zeilennummern im Befehlsfenster.

Um die Option "Tron" wieder abzuschalten, halten Sie Ihr Programm an, öffnen das Menü "Debug" und klicken auf die Option "Troff". Bestätigen Sie auch hier Ihre Wahl durch Klicken in das "Ok"-Feld im Dialogfeld.

TRACE und TRON werden in Anhang C noch ausführlich beschrieben.

Speichern von Programmen
Um ein Programm auf Diskette zu speichern, öffnen Sie das Menü "File" und klicken auf die Option "Save As".



Geben Sie ZAEHL in das Auswahlfeld (ITEM SELECTOR) ein. Beachten Sie, daß von ST BASIC automatisch der Extender .BAS an Ihren Dateinamen angefügt wird. Durch den Extender wird ST BASIC kenntlich gemacht, daß es sich bei der Datei um eine ST BASIC-Programmdatei handelt. Um die Datei auf Diskette zu speichern, klicken Sie in das "Ok"-Feld. Sobald die Bestätigung auf dem Bildschirm erscheint, ist Ihre Datei auf der Diskette abgelegt.

Sie können ein Programm auch durch Eingeben von

SAVE ZAEHL [RETURN]

speichern. ST BASIC legt diese Datei als ZAEHL.BAS auf Diskette ab.

Anmerkung: Die Option "Save As" ersetzt (überschreibt) jede Datei auf der Diskette, die unter demselben Dateinamen abgelegt wurde. Wenn Sie "SAVE" im Befehlsfenster eintippen, wird im Gegensatz dazu eine gegebenenfalls bereits bestehende Datei mit demselben Namen nicht gelöscht.

Laden von Programmen
Geben Sie NEW [RETURN] ein, um das im Arbeitsspeicher befindliche Programm zu löschen. Vergewissern sie sich durch Eingeben von LIST, daß der Arbeitsspeicher frei ist.

Um das zuvor gespeicherte Programm von Diskette in den Arbeitsspeicher zu laden, öffnen Sie das Menü "File" und klicken auf die Option "Load". Im Auswahlfeld erscheint die Angabe ZAEHL.BAS. Sie können ZAEHL.BAS durch Klicken auf den Dateinamen und nachfolgendes Klicken in das "Ok"-Feld auswählen. Sobald die Anfrage "Ok" auf dem Bildschirm erscheint, befindet sich Ihr Programm im Speicher. Um sich hiervon zu vergewissern, können Sie es über LIST auflisten lassen. Die Kopfzeile, "List of ZAEHL.BAS" dient als Hinweis für Sie, daß das aufgelistete Programm unter dem Namen ZAEHL.BAS abgelegt ist.

Sie können ein Programm auch durch Eingeben von

LOAD ZAEHL

in den Arbeitsspeicher des Computers laden.

Verknüpfen von Programmen
Manchmal ist es einfacher und bequemer, ein Programm in einzelnen Modulen zu schreiben und diese zu einem späteren Zeitpunkt zu assemblieren. Diese Möglichkeit besteht über die Funktion MERGE.

Geben Sie das nachstehende Programm ein und speichern Sie es unter dem Namen NACHTRAG.BAS.

```
20 PRINT "VERLAENGERT DURCH VERKNUEPFEN"
30 END
```

Geben Sie NEW ein und schreiben Sie dieses Programm:

```
10 PRINT "DAS IST EIN KURZES PROGRAMM"
20 END
```

Geben Sie RUN ein, um das Programm ablaufen zu lassen.

Wählen Sie dann die Option "Merge" aus dem Menü "File" aus. Daraufhin klicken Sie auf "NACHTRAG.BAS" im Auswahlfeld und bestätigen Ihre Wahl über das "Ok"-Feld.

Listen sie Ihr Programm auf. Wie Sie sehen können, wurden die beiden Programmsegmente verbunden. Wenn Sie die Zeile 20 betrachten, werden Sie feststellen, daß die Zeile 20 des ersten Programmes, END, durch Zeile 20 des damit verknüpften Programmes ersetzt wurde. Aus diesem Grund sollten Sie Ihre Zeilennumerierung sehr sorgfältig vornehmen, wenn sie mit der Option MERGE arbeiten.

Löschen von Programmen
Um ein Programm zu löschen, klicken Sie auf die Option "Delete File" im Menü "File". Wählen Sie durch Klicken den Namen der zu löschenden Datei, wie beispielsweise NACHTRAG.BAS, aus und bestätigen Sie diese Wahl durch Klicken in das "Ok"-Feld. Sobald die Anfrage "Ok" auf dem Bildschirm erscheint, ist diese Datei gelöscht.

Beenden des Programmes ST BASIC
Um die Arbeit mit ST BASIC zu beenden, öffnen Sie das Menü "File" und klicken hier auf die Option "Quit".

Befehlseingabe über Tastatur
Sofern Sie dies vorziehen, können Sie Programmierbefehle über die ST Tastatur eingeben, anstatt sie unter Verwendung der Maus auszuwählen. Diese abgekürzten Eingaben sind:

AUTO
[CONTROL][G] Hält ein Programm an oder beendet die automatische Zeilennumerierung
[CONTROL][C] Hält ein Programm an und beendet es, ohne eine Möglichkeit zu bieten, mit diesem Programm fortzufahren

CONT oder [RETURN] (um mit einem Programmlauf fortzufahren)
DELETE <Zeilennummern-Liste> (um Programmzeilen zu löschen)
EDIT oder ED (um in den Editier-Modus zu gelangen)
ERA <Dateiname> (um eine Datei zu löschen)
LOAD <Dateiname> (um eine Datei zu laden)
MERGE <Dateiname> (um Programme zu verknüpfen)
NEW (um den Arbeitsspeicher zu löschen)
QUIT (um ST BASIC zu beenden)
RUN <Dateiname> (um ein Programm ablaufen zu lassen)
SAVE <Dateiname> (um ein Programm zu speichern)
STEP (um ein Programm schrittweise zu durchlaufen)
TRACE (um die Fehleraufdeckung einzuschalten)
TROFF (um die Zeilennummern-Anzeige abzuschalten)
TRON (um die Zeilennummern-Anzeige einzuschalten)
UNTRACE (um die Fehleraufdeckung abzuschalten)

Eine vollständige Auflistung aller ST BASIC-Befehle finden sie in Anhang A dieses Handbuches.

Gespeicherte Grafiken
Um gespeicherte Grafiken mit ST BASIC auf Ihrem ST Computer-System verwenden zu können, müssen Sie Platz im Arbeitsspeicher schaffen.

Wenn Sie auf das Schreibtischzubehör des GEM-Desktop verzichten, können Sie 30.000 Bytes Speicherplatz gewinnen. Um das Schreibtischzubehör zu entfernen, können sie zwei Methoden anwenden:

1. Löschen Sie das Schreibtischzubehör auf dem Sicherungsduplikat der ST BASIC Programmdiskette. Dazu öffnen Sie einfach das Diskettenfenster der Programmdiskette und werfen die Datei DESK.ACC in den Papierkorb. In diesem Fall haben Sie das Schreibtischzubehör noch immer auf Ihrer Originaldiskette gespeichert und können es von dort wieder auf das Sicherungsduplikat zurückkopieren und verwenden.

2. Geben Sie der Datei DESK.ACC einen neuen Namen. Dann können Sie die Datei DESK.ACC auswählen, das Menü "File" öffnen und die Option "Show Info" auswählen. Im daraufhin erscheinenden Dialogfeld steht ein Cursor am Ende des Dateinamens. Durch Drücken der [Backspace]-Taste kann der Dateiname DESK.ACC nun gelöscht werden. Verwenden sie als neuen Namen eine beliebige Bezeichnung. Lediglich der Extender darf nicht auf .ACC lauten.

Anmerkung: Genauere Informationen über das Löschen und Umbenennen von Dateien erhalten Sie im ATARI ST Bedienungshandbuch.

ERWEITERUNG DES ARBEITSSPEICHERS FÜR ST BASIC

Nachdem Sie TOS von der Systemdiskette, und ST BASIC von der ST Programmdiskette in den Arbeitsspeicher des ST Computers geladen haben, verbleibt lediglich ein geringfügiger Teil des Speicherplatzes für Ihre Programmierung.

Für die Erweiterung des verfügbaren Speicherplatzes stehen Ihnen zwei Möglichkeiten zur Verfügung:

1. Schalten Sie die Option "Buffer Graphics" ab. Damit gewinnen Sie zusätzliche 32.000 Bytes an verfügbarem Speicherplatz. Zeigen Sie hierzu auf das Menü "Run" und kontrollieren Sie, ob sich vor der Option "Buf Graphics" ein Häkchen befindet. In diesem Fall wählen Sie die Option aus und klicken in dem daraufhin erscheinenden Dialogfeld in das "Ok"- Feld, um die Option abzuschalten.

Anmerkung: Falls Sie die Option "Buf Graphics" abschalten, während sich ein Programm im Arbeitsspeicher befindet, wird dieses Programm im Speicher gelöscht.

2. Wenn Sie auf die Verwendung des Schreibtischzubehörs verzichten können, können Sie über weitere 30.000 Bytes an Speicherplatz für Ihre Programmierarbeiten verfügen. Lesen Sie hierzu auf Seite 16 dieses Handbuches nach.

ANHANG A

RESERVIERTE WÖRTER IN ST BASIC

Nachfolgend sehen Sie eine Auflistung aller reservierten Wörter, die in ST BASIC verwendet werden. Falls Sie eines dieser Wörter als Variablennamen benutzen, erscheint die Fehlermeldung "something is wrong". Eine detaillierte Beschreibung aller reservierten Wörter erhalten Sie in Anhang C.

ABS
ALL
AND
AS
ASC
ATN
AUTO
BASE
BLOAD
BREAK
BSAVE
CALL
CDBL
CHAIN
CHR$
CINT
CIRCLE
CLEAR
CLEARW
CLOSE
CLOSEW
COLOR
COMMON
CONT
CONTRL
CONT
COS
CSNG
CVD
CVI
CVS
DATA
DEF
DEF FN
DEF SEG
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
DIR
EDIT
ELLIPSE
ELSE
END
EOF
EQV
ERA
ERASE
ERL
ERR
ERROR
EXP
FIELD
FIELD#
FILL
FIX
FLOAT
FOLLOW
FOR
FRE
FULLW
GB
GEMSYS
GET
GET#
GO
GOSUB
GOTO
GOTOXY
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT$
INPUT$
INSTR
INT
INTIN
INTOUT
KILL
LEFT$
LEN
LET
LINE INPUT
LINE INPUT#
LINEF
LIST

LLIST
LOAD
LOC
LOF
LOG
LOG10
LPOS
LPRINT
LSET
MERGE
MID$
MKD$
MKI$
MKS$
MOD
NAME
NEW
NEXT
NOT
OCT$
OLD
ON
OPEN
OPENW
OPTION
OR
OUT
PCIRCLE
PEEK
PELLIPSE
POKE
POS
PRINT
PRINT#
PRINT USING
PTSIN
PTSOUT
PUT
QUIT
RANDOMIZE
READ
REM
RENUM
REPLACE
RESET
RESTORE
RESUME
RETURN
RIGHT$
RND
RSET
RUN
SAVE
SGN
SIN
SOUND
SPACE$
SPC
SQR
STEP
STOP
STR$
STRING$
SWAP
SYSDBG
SYSTAB
SYSTEM
TAB
TAN
THEN
TO
TRACE
TROFF
TRON
UNBREAK
UNFOLLOW
UNTRACE
USING
VAL
VARPTR
VDISYS
WAIT
WAVE
WEND
WHILE
WIDTH
WRITE
WRITE#
XOR

ANHANG B

LOGISCHE OPERATOREN, VORRANGREGELN UND FUNKTIONEN VON ST BASIC

LOGISCHE OPERATOREN

Die von ST BASIC anerkannten Operatoren sind NOT, AND, OR, XOR, IMP und EQV. Diese logischen Operatoren arbeiten auf der Basis von Flags, die aus logischen Ausdrücken resultieren. Eine TRUE-Flag entspricht -1, eine FALSE-Flag 0. Deshalb ergibt die Anweisung "A=1: B=2: PRINT A=B" den Wert 0, während die Anweisung "A=1: B=2: PRINT A<>B" das Resultat -1 ergibt.

Das Ergebnis von AND ist TRUE, wenn beide Argumente wahr sind.
Beispiel: 2+2=4 AND 3+2=5 ergibt TRUE.

Das Ergebnis von OR ist TRUE, wenn eines der Argumente wahr ist.
Beispiel: 2+2=4 OR 3+2=7 ergibt TRUE.

IMP ist die Abkürzung für IMPLICATION (Folgerung). IMP arbeitet auf der Basis logischer Ausdrücke und überprüft die Gültigkeit von Prämissen und Folgerungen. IMP ist in allen Fällen gültig, es sei denn, eine Prämisse ergibt TRUE, die Folgerung dagegen FALSE.

Die Anweisung "2+2=4 IMP 3+2=6" ergibt FALSE.

Die nachfolgenden Anweisungen sind gültige Folgerungen und ergeben TRUE:

2+2=4 IMP 3+3=6
2+2=3 IMP 3+3=6
2+2=3 IMP 3+3=7

Die nachstehenden Operatoren arbeiten bitweise mit Ein-Byte Integerwerten wie folgt:

AND ergibt ein Resultat, in dem ein Bit nur dann 1 entspricht, wenn beide Argumente eine 1 enthalten. So ergibt "A%=5: B%=3: C%=A% AND B% für C% den Wert 1.

OR produziert ein Ergebnis, in dem ein Bit 1 entspricht, wenn eines der Argumente eine 1 enthält. Beispielsweise entspricht in "A%=5: B%=3: C%=A% OR B%" C% dem Wert 7.

XOR erstellt Ergebnisse, in denen ein Bit dann 1 entspricht, wenn lediglich ein Argument eine 1 enthält. In diesem Fall ergibt "A%=5: B%=3: C%=A% XOR B%" für C% den Wert 6.

EQV ergibt Resultate, in denen ein Bit 1 entspricht, wenn entweder in beiden Argumenten eine 1, oder in beiden Argumenten eine 0 enthalten ist. Ein Bit entspricht dann 0, wenn die Bits in den beiden Argumenten unterschiedlich sind. Demnach ergibt "A%=5: B%=3: C%=A% EQV B%" den Wert -7 für C%.

TRUE-Tabelle für logische Operationen

NOT

X	NOT X
0	1
1	0

AND

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

IMP

X	Y	X IMP Y
0	0	1
0	1	1
1	0	0
1	1	1

EQV

X	Y	X EQV Y
0	0	1
0	1	0
1	0	0
1	1	1

Arithmetische Operatoren

Symbol	Name	Beispiel
+	Addition	X + Y
−	Subtraktion	X − Y
*	Multiplikation	X * Y
/	Division	X / Y
	Integer-Division	X Y
MOD	Modul	X MOD Y
^	Exponierung	X ^ Y

Relationale Operatoren

Symbol	Bedeutung	Beispiel
=	Gleichzeichen	X = Y
<>	Ungleichzeichen	X <> Y
<	Kleiner als - Zeichen	X < Y
>	Größer als - Zeichen	X > Y
<=	Kleiner Gleich - Zeichen	X <= Y
>=	Größer Gleich - Zeichen	X >= Y

Vorrangregeln der Operatoren

Operator	Erklärung
( )	Begriffe in Klammern haben höchste Priorität
^	Exponierung
–	Negatives Vorzeichen
*	Multiplikation
/	Fließkomma- und Integer-Division
MOD	Modul
+, –	Addition, Subtraktion
=, <>, <, >, <=, >=	Relationale Operatoren
NOT, AND, OR, XOR	Logische Operatoren in der angegebenen
IMP, EQV	Reihenfolge

Zusammenfassung der Funktionen von ST BASIC

Funktionen arbeiten mit Konstanten und Variablen, um Werte für Variablen zu erhalten. Eine Konstante ist eine Zahl wie beispielsweise 250.4 oder eine Zeichenkette wie z.B. "HALLO". Eine Variable ist ein bezeichneter numerischer Wert wie GESAMT oder ein bezeichneter String wie NAME$.

Variablennamen

Variablennamen dürfen keine Leerzeichen enthalten. Sie dürfen beliebig lang sein. Allerdings ist zu beachten, daß lediglich die ersten 31 Zeichen einer Variablen als Unterscheidungsmerkmal von ST BASIC verwendet werden.

Numerische Variablen

Numerische Variablen gibt es in unterschiedlichen Formen. In der nachstehenden Tabelle finden sie eine Zusammenstellung von Variablentypen.

ZEICHEN FÜR DIE VARIABLENDEKLARATION

Zeichen	Typ	Beispiel
$	String	NAME$
%	Integerzahl	DATEN.NUMMER%
!	Realzahl	GESAMT.GEWINN!

Typendeklaration

Die nachfolgenden Anweisungen deklarieren Variablentypen in ST BASIC. (Lesen Sie hierzu auch in Anhang C nach.)

DEFSTR bezeichnet String-Variablen
DEFINT bezeichnet Integer-Variablen
DEFSNG bezeichnet Realzahlen-Variablen

Numerische Funktionen
Die in ST BASIC verfügbaren numerischen Funktionen können Sie der nachstehenden Tabelle entnehmen.

NUMERISCHE FUNKTIONEN

Funktion	Erklärung
ABS	Gibt den absoluten Wert einer Zahl aus
ATN	Gibt den Arcustangens einer Zahl aus
COS	Gibt den Cosinus einer Zahl aus
EXP	Gibt die Potenz e eines Wertes aus
LOG	Gibt den natürlichen Logarithmus einer Zahl aus
LOG10	Gibt den Basis 10 - Logarithmus einer Zahl aus
RND	Generiert eine Folge zufällig gewählter Zahlen
SIN	Gibt den Sinus einer Zahl in Radian aus
SQR	Gibt die Quadratwurzel einer Zahl aus
TAN	Gibt den Tangens einer Zahl in Radian aus

String-Funktionen
Strings können mit Pluszeichen (+) verkettet werden (z.B. A$ = B$ + C$). In der nachstehenden Tabelle sehen Sie die anderen in ST BASIC verfügbaren String-Funktionen.

STRING-FUNKTIONEN

Funktion	Erklärung
INSTR	Sucht das erste Vorkommen einer bestimmten Zeichenfolge in einem String und gibt deren Position aus.
LEFT$	Gibt die ersten Zeichen einer Zeichenkette aus.
LEN	Gibt die Zeichenlänge eines Strings aus.
MID$	Entfernt einen String aus einer Zeichenkette.
RIGHT$	Gibt die letzten Zeichen eines Strings aus.
SPACE$	Gibt einen String aus, der aus Leerzeichen besteht.
STR$	Wandelt eine Zahl in einen String um.
STRING$	Gibt einen String mit der angegebenen Länge aus.

Arrays
ST BASIC verfügt über numerische und String-Arrays. Über die Anweisung DIM werden Variablen dimensioniert. Im Bezug auf Arrays beziehen sich Unterbereiche auf Reihen, Spalten und Ebenen - in dieser Reihenfolge. Werte von Unterbereichen können jede gültige numerische Konstante, Variable oder beliebige numerische Ausdrücke sein. Am effizientesten ist die Verwendung von Integerzahlen, da bei der Angabe von Realzahlen diese für eine Verwendung als Unterbereich in einem Array zuvor in Integerzahlen umgewandelt werden. Arrays akzeptieren direkte Eingaben und können wie jede andere Variable in einer BASIC-Anweisung verwendet werden.

ZWEIDIMENSIONALES ARRAY

	(0)	(1)	(2)	
(0)	(0,0)	(0.1)	(0.2)	SO
(1)	(1.0)	(1.1)	(1.2)	MO
(2)	(2.0)	(2.1)	(2.2)	DI
(3)	(3.0)	(3.1)	(3.2)	MI
(4)	(4.0)	(4.1)	(4.2)	DO
(5)	(5.0)	(5.1)	(5.2)	FR
(6)	(6.0)	(6.1)	(6.2)	SA
	6 AM	2 PM	10 PM	

Die maximale Anzahl von Elementen in einem Array ist durch den verfügbaren Speicherplatz eingeschränkt. Elemente unterschiedlicher Datentypen belegen unterschiedlich viel Speicherplatz.

INTEGERZAHLEN belegen 2 Bytes
REALZAHLEN belegen 4 Bytes
STRINGS belegen 6 Bytes.

Zeilenformat
Das Zeilenformat in ST BASIC ist:

<Zeilennummer> <Sprungmarke:> <Anweisung> <:Anweisung> <:'REM>

Die Angabe einer Sprungmarke ist optional. Eine Sprungmarke kann anstelle einer Zeilennummer als Zeilenangabe innerhalb einer GOTO- oder GOSUB-Anweisung verwendet werden.

Einschränkungen bei der Angabe des Dateinamens
ST BASIC Programmzeilen verwenden den Extender .BAS, um als BASIC-Programme gekennzeichnet zu sein. Ein Dateiname darf nicht mehr als 8 Zeichen, und der Extender nicht mehr als 3 Zeichen umfassen. Der Dateiname STBASIC.DAT ist beispielsweise eine gültige Angabe.

ANHANG C

BEFEHLE, FUNKTIONEN UND ANWEISUNGEN

In diesem Abschnitt werden die Befehle, Funktionen und Anweisungen von ST BASIC in alphabetischer Reihenfolge aufgeführt und beschrieben. Die Schreibweise der Syntax-Angaben hat folgende Bedeutung:

● ● ● Wörter in spitzen Klammern, < >, beschreiben die Datenart, die Sie hier eingeben müssen. Diese Angaben erklären sich selbst. So bedeutet beispielsweise <Variable>, daß Sie an dieser Position innerhalb einer Anweisung eine Variable vorgeben müssen.

● ● ● Begriffe in eckigen Klammern, [ ], können optional verwendet werden, dürfen jedoch nicht mehrmals hintereinander geschrieben werden.

● ● ● Begriffe in runden Klammern, ( ), können optional verwendet werden. Im Gegensatz zu Angaben in eckigen Klammern können diese Begriffe mehrmals hintereinander geschrieben werden.

● ● ● In Großbuchstaben geschriebene Wörter sind ST BASIC Befehlswörter.

ABS **X=ABS(N)** **FUNKTION**

Syntax: X = ABS (<numerischer Ausdruck>)

Effekt: Gibt den absoluten Wert einer Zahl aus. Der absolute Wert einer Zahl ist immer positiv oder Null.

Erklärung:

ABS gibt einen Integer-Wert für ein Integer-Argument aus. Bei Realzahlen hat der ausgegebene Wert dieselbe Genauigkeit wie das Argument.

Beispiel:

```
Ok 10 I% = ABS (-9)
Ok 20 PRINT I%
Ok 30 X! = ABS(325556.244)
OK 40 PRINT X!
Ok 50 END
Ok RUN
  9
  325556
Ok
```

ABS **X=ABS (N)** **FUNKTION**

Syntax: X = ABS (<numerischer Ausdruck>)

Effekt: Gibt den absoluten Wert einer Zahl aus. Der absolute Wert einer Zahl ist immer positiv oder Null.

Erklärung:

ABS gibt einen Integer-Wert für ein Integer-Argument aus. Bei Realzahlen hat der ausgegebene Wert dieselbe Genauigkeit wie das Argument.

Beispiel:

```
Ok 10 I% = ABS (-9)
Ok 20 PRINT I%
Ok 30 X! = ABS(325556.244)
OK 40 PRINT X!
Ok 50 END
Ok RUN
  9
  325556
Ok
```

ATN **X!=ATN (N%)** **FUNKTION**

Syntax: X! = ATN (<numerischer Ausdruck>)

Effekt: Errechnet den Arcustangens einer Zahl.

Erklärung:

Die Funktion ATN gibt eine Realzahl mit einfacher Genauigkeit aus. Die Zahl ist ein Winkel in Radian, der zwischen -PI/2 und PI/2 liegt. Die Funktion TAN ist das Gegenstück zu ATN.

Beispiel:

```
Ok 10 RADIAN! = ATN (0.99999)
Ok 20 PRINT "Der Winkel in Radian ist ";RADIAN!
Ok 30 PRINT
Ok 40 PI = 3.14159
Ok 50 GRAD = RADIAN! * 180/PI
Ok 60 PRINT "Der Winkel in Grad ist "=;CINT(GRAD)
Ok RUN
Der Winkel in Radian ist .785393
Der Winkel in Grad ist 45
Ok
```

Syntax: AUTO [<erste Zeilennummer>] [,<Erhöhung>]

Effekt: Erstellt für jedes Betätigen der RETURN-Taste eine Zeilennummer. Mit [CTRL] [G] wird AUTO abgeschaltet. Eine Zeilennummer darf nicht größer als 65535 sein. Der Befehl AUTO kann innerhalb des Editors nicht verwendet werden.

Erklärung:

Sie geben die erste zu erstellende Zeilennummer, sowie die Erhöhung zur nächsten zu generierenden Zeilennumer an. Wenn Sie die erste zu erstellende Zeilennummer nicht angeben, beginnt AUTO mit Zeilennummer 10. Geben Sie keine Erhöhung vor, erfolgt die Erhöhung entweder in Zehnersprüngen oder in der zuletzt durch einen AUTO-Befehl angegebenen Weise.

Existiert eine Zeilennummer bereits, druckt AUTO zwei Sternchen vor diese Nummer (**10). Wenn Sie eine neue Programmzeile eingeben, wird die alte Zeilennummer nach Betätigen der RETURN- Taste durch die neue Zeile ersetzt. Drücken Sie lediglich die RETURN-Taste, bleibt die alte Zeilennummer unverändert.

Mit [CTRL] [G] wird AUTO abgeschaltet. [CTRL] [G] entspricht von der Funktion her nicht der RETURN-Taste. [CTRL] [G] gibt keine Programmzeile ein und verändert eine bereits vorhandene Zeile nicht.

Beispiele:

```
Ok AUTO                 Ok AUTO 50, 25
10                       50
20                       75
30                      100
 .                        .
 .                        .
 .                        .

Ok AUTO , 20            Ok AUTO 50
10                       50
30                       70
50                       90
 .                        .
 .                        .
 .                        .
```

Syntax: BLOAD <Dateiangabe,>[,<Adresse>]

Effekt: Lädt eine Datei in den Arbeitsspeicher.

Erklärung:

BLOAD wird verwendet, um Maschinensprache-Programme und Arrays sowie deren Inhalt zu laden. BLOAD kann zudem Bildschirmbilder darstellen.

BLOAD lädt eine Datei an der von Ihnen angegebenen Adresse in den Arbeitsspeicher. Die Dateiangabe ist der vollständige Dateiname inclusive Extender. Die Adresse ist der numerische Ausdruck, bei dem mit dem Laden begonnen werden soll.

Wenn Sie die Adresse nicht angeben, wird der mit BSAVE spezifizierte Startpunkt angenommen. Die Datei wird an dieselbe Adresse geladen, an der sie sich zuvor befand.

BLOAD überprüft keine Adressen. Obwohl es möglich ist, an jeder beliebigen Stelle die Anweisung BLOAD zu verwenden, sollten Sie sie nicht über Datenbereiche von BASIC oder über Ihr Programm laden. In diesem Fall könnte Ihr Programm abstürzen.

Beispiel:

```
Ok 110 BLOAD "ARRAY",23
```

Syntax: BREAK [<Zeilennummern-Liste>]

Effekt: Hält den Programmlauf an.

Erklärung:

BREAK, alleine verwendet, bewirkt, daß der Programmlauf nach Ausführung jeder Programmzeile angehalten wird. Sowohl die Programmzeile, als auch jede eventuelle Ausgabe wird ausgegeben. Durch Betätigen von [RETURN] oder Eingeben von CONT wird die nächste Programmzeile ausgeführt. Damit entspricht BREAK dem Befehl STEP.

Geben Sie hinter BREAK Zeilennummern an, hält das Programm nur in den betreffenden Zeilen an.

Der BREAK-Modus wird durch den Befehl UNBREAK abgeschaltet.

Beispiel:

```
Ok 10 N = 5
Ok 20 FOR X = 1 TO 5
Ok 30 N = N - 1
Ok 40 PRINT N
Ok 50 NEXT X
Ok BREAK 50
Ok RUN
  4
b 50 NEXT X
Br
```

Syntax: BSAVE <Dateiangabe>,<Adresse>,<Länge>

Effekt: Legt einen Teil des Arbeitsspeichers in einer Datei ab.

Erklärung:

BSAVE speichert Maschinensprache-Programme, Daten oder Bildschirmbilder ab.

<Dateiangabe> ist die Angabe Ihres Dateinamens.

Die Adresse ist ein numerischer Ausdruck.

Beispiel:

```
Ok BSAVE "ARRAY" ,23,650
```

Syntax: CALL <numerischer Variable> [(<Parameter Liste>)]

Effekt: Gibt die Programmkontrolle an eine Maschinensprache-Unterroutine ab.

Erklärung:

Die numerische Variable ist die Anfangs-Speicheradresse der Maschinensprache-Unterroutine. Die Routine wird mit BLOAD in den Arbeitsspeicher geladen.

Die optionale Parameter-Liste besteht aus Ausdrücken, die als Argumente für die Datenübertragung zwischen Hauptprogramm und Assembler-Routine dienen. Die Parameter-Liste wird in Klammern eingeschlossen und muß durch Kommata abgegrenzt werden.

Beispiel:

```
Ok 500 BLOAD "ASHLER",185000
Ok 550 CHART = 185666
Ok 600 CALL CHART(I%, A$, X)
```

Anmerkung: Die Assembler-Routine, die über den Befehl CALL aufgerufen wird, sucht nach 2 Parametern im Anwender-Stack (A7). Der erste Parameter ist eine 2-Byte-Integerzahl, die die Anzahl formaler Parameter angibt, die vom Anwenderprogramm übertragen wurden (in Zeile 600 im obigen Beispiel wäre die Anzahl der Parameter 3). Jeder dieser Werte belegt 8 Bytes im Array, unabhängig davon, ob es sich bei dem formalen Parameter um eine Integerzahl, Realzahl oder andere Werte handelt. In jedem Fall enthält bei Verwendung einer String-Variablen als formaler Paramater der 8-Byte-Wert im Array einen Zeiger zur Speicheradresse, in der dieser String abgelegt ist.

CHAIN **CHAIN NEUPROG, 100, ALL** **ANWEISUNG**
CHAIN MERGE NEUPROG, 100 DELETE 500-600

Syntax: CHAIN <Dateiname>[,<Zeilenangabe>][,ALL]
CHAIN MERGE <Dateiname>[,Zeilenangabe>]
[,DELETE<Zeilenangabe-Liste>]

Effekt: Übergibt die Kontrolle und überträgt Variablen an ein anderes Programm. Der Extender .BAS wird automatisch an den Dateinamen angesetzt, sofern Sie nicht selbst einen anderen Extender vorgeben.

Erklärung:

Das Programm, das Sie über die CHAIN-Anweisung spezifizieren, ersetzt das derzeitige Programm im Arbeitsspeicher. Das mit CHAIN eingebrachte Programm wird oft auch Overlay genannt, da es das ursprüngliche Programm oder Teile davon überschreibt. <Dateiname> ist der Dateiname des neuen Programmes. Es kann sich um jeden beliebigen String-Ausdruck eines legalen Dateinamens handeln.

Die Option MERGE verknüpft ein ein Programm mit einem existenten Programm, anstatt es zu ersetzen. CHAIN MERGE speichert alle Variablen, Typendeklarationen, Anweisungen und Optionen. Wenn Sie die Option MERGE weglassen, müssen Sie alle DEF-Anweisungen in jedem mit CHAIN neu angefügten Programm neu aufstellen. Die Option MERGE ersetzt die Anweisungen im neuen Programm durch die Anweisungen im ursprünglichen Programm. Sind einige der Zeilennummern im neuen Programm identisch zu Zeilennummern im ursprünglichen Programm, ersetzen die neuen Programmzeilen die alten.

Sie können nach dem Dateinamen eine Zeilenangabe spezifizieren, durch die angezeigt wird, ab welcher Zeile im neuen Programm mit der Programmausführung begonnen werden soll. Andernfalls beginnt der Programmlauf mit der ersten ausführbaren Anweisung.

Die Option ALL zeigt an, daß alle Variablen aus dem ursprünglichen Programm an das neue Programm übertragen werden. ALL in Verbindung mit CHAIN MERGE ist nicht gültig.

Wird die Option ALL weggelassen, müssen Sie mit der Anweisung COMMON angeben, welche Variablen sowohl von dem ursprünglichen, als auch vom neuen Programm verwendet werden können.

Lesen Sie hierzu auch unter COMMON nach.

Verwenden Sie die Option DELETE nur in Verbindung mit CHAIN MERGE. Mit DELETE können Sie Teile des alten Programmes aus dem Arbeitsspeicher entfernen, um Platz für das neue Programm zu schaffen. Die Option DELETE löscht Zeilen aus dem alten Programm, bevor es mit dem neuen Programm <Dateiname> verbunden wird. Geben Sie nach dem Befehlswort DELETE die Zeilennummern an, die gelöscht werden sollen.

Beispiele:

Die nachfolgende Anweisung bindet ein Programm mit dem Namen KURS.BAS ein:

```
Ok 400 CHAIN "KURS"
```

In diesem Beispiel wird das Programm KURS.BAS eingebunden. Der Programmlauf beginnt bei Zeile 1200. Alle Programm-Variablen können vom ursprünglichen Programm in das neue Programm übernommen werden.

```
Ok 400 CHAIN "KURS", 1200, ALL
```

Im letzten Beispiel werden die Zeilen eines Overlays mit dem Namen GESAMT.OVR mit dem bereits im Arbeitsspeicher befindlichen Programm verknüpft. Die Programmausführung beginnt bei Zeile 900. Bevor die verknüpfte Datei in den Speicher geladen wird, wird durch die Anweisung eine Zeilenangabe-Liste, die von Zeile 900 bis zu Zeile 2000 reicht, gelöscht.

```
Ok 710 CHAIN MERGE "GESAMT.OVR", 900, DELETE 900-2000
```

Syntax: A$ = CHR$(<numerischer Ausdruck>)

Effekt: Gibt das Zeichen aus, dessen ASCII-Wert der eingegebenen Dezimalzahl entspricht.

Erklärung:

CHR$ gibt einen Ein-Zeichen-String aus.

Der numerische Ausdruck muß einer legalen Integerzahl entsprechen.

Der ASCII-Wert des ausgegebenen Zeichens ist <Ausdruck>MOD 256. Das bedeutet, daß der Ausdruck in eine Zahl zwischen 0 und 256 konvertiert wird. Ist der Ausdruck größer als 256, wird er wie ein Restwert einer Division durch 256 behandelt (siehe Beispiele).

CHR$ wandelt Realzahlen in Integerzahlen um.

Verwenden Sie die Funktion CHR$, um Sonderzeichen wie Zeilenvorschübe oder Zeilenschaltungen an einen Ausgabe- Datenkanal zu übermitteln.

CHR$ ist das Gegenstück zu ASC.

Beispiel:

```
Ok 10 PRINT CHR$(83)
Ok 20 PRINT CHR$(100)
Ok 30 PRINT CHR$(356)
Ok RUN
S
d
d
Ok
```

Syntax: I% = CINT(<numerischer Ausdruck>)

Effekt: Rundet eine Zahl auf die nächste Integerzahl auf oder ab.

Erklärung:

Der numerische Ausdruck muß zwischen -32768 und 32767 liegen. Andernfalls tritt ein Überlauffehler auf.

Lesen Sie auch unter FIX und INT nach.

Beispiel:

```
Ok 10 PRINT CINT(5.2)
Ok 20 PRINT CINT(62.89)
Ok 30 PRINT CINT(-456.61)
Ok RUN
  5
  63
 -457
Ok
```

Syntax: CIRCLE <horizont. Mittelpunkt, vertik. Mittelpunkt, Radius>[<,Anfangswinkel, Endwinkel>]

Effekt: CIRCLE zeichnet Kreise und Kreisausschnitte.

Erklärung:

CIRCLE zeichnet einen Kreis, dessen Mittelpunkt an dem Punkt liegt, der durch die ersten beiden Parameter (horizontaler und vertikaler Mittelpunkt) vorgegeben wurde. Die Positionen werden in Pixel angegeben und von der oberen linken Ecke des Ausgabefensters aus gezählt.

Der dritte Parameter (Radius) wird ebenfalls in Pixel ausgedrückt. Die horizontale und vertikale Pixelangabe ist abhängig von der gewählten Auflösung, sowie von der Größe des Ausgabefensters. Der Kreis wird in der festgelegten Zeichenfarbe (Parameter 3 der COLOR-Anweisung) dargestellt.

Die letzten beiden Parameter (Anfangs- und Endwinkel) sind optional. Wird hier nichts angegeben, zeichnet CIRCLE einen Kreis. Andernfalls zieht CIRCLE einen Kreisausschnitt zwischen den beiden Punkten. CIRCLE zeichnet allerdings nur einen Kreisbogen und kein eingefärbtes Kreissegment. Winkel werden in Grad mal 10 angegeben. So werden 45 Grad als 450, 180 Grad als 1800 usw. angegeben. O Grad zeigt zum rechten Rand des Fensters, 90 Grad zum oberen, 180 Grad zum linken und 270 Grad zum unteren Fensterrand. CIRCLE 100,30,30,0,3600 zeichnet einen vollständigen schwarzen Kreis.

Lesen Sie auch unter PCIRCLE, ELLIPSE und PELLIPSE nach.

Beispiel:

```
Ok 10 COLOR 1,0,1: CLEARW 2
Ok 20 CIRCLE 100,50,40
Ok 30 COLOR 1,0,2
Ok 40 CIRCLE 100,50,40,300,90
Ok RUN
[Im Ausgabefenster erscheint ein schwarzer Kreis mit einem roten Kreisausschnitt über 60 Grad, beginnend bei 30 Grad]
Ok
```

Syntax: CLEAR

Effekt: Bereinigt den mit Programmdaten belegten Arbeitsspeicher, ohne das derzeit im Arbeitsspeicher befindliche Programm zu löschen.

Erklärung:

CLEAR setzt alle numerischen Variablen und String-Variablen auf Null. Der Befehl CLEAR macht alle Arrays undefiniert.

Beispiele:

Das nachfolgende Beispiel löscht alle Daten aus dem Speicher, ohne das Originalprogramm zu löschen:

```
Ok CLEAR
```


Syntax: CLEARW <numerischer Ausdruck>

Effekt: CLEARW löscht BASIC-Fensterinhalte.

Erklärung:

CLEARW löscht den Inhalt des angegebenen Fensters. Dabei werden die Fenster folgendermaßen bezeichnet:

0 = Bearbeitungsfenster
1 = Auflistungsfenster
2 = Ausgabefenster
3 = Befehlsfenster

Beispiel:

```
Ok 10 CLEARW 2
Ok 20 PRINT "HALLO"
Ok RUN
```

CLOSE #1
CLOSE 1,3,4

Syntax: CLOSE [#]<Dateinummer>

Effekt: Schließt geöffnete Disketten-Dateien und schließt jede Ein- oder Ausgabe ab.

Erklärung:

Die CLOSE-Anweisung schließt geöffnete Dateien, beläßt die Dateinummern unverändert und gibt den gesamten Speicherplatz frei, der von der Datei verwendet wird. Die Dateien müssen unter Verwendung der OPEN-Anweisung geöffnet worden sein.

Die Dateinummer ist die Kennummer, die Sie einer Datei in der OPEN-Anweisung zuordnen. Sie können eine beliebige Anzahl von Dateinummern in der optionalen CLOSE-Anweisung angeben. Trennen Sie hierbei die einzelnen Dateinummern durch Kommata ab.

Ein Nummernzeichen (#) vor der Dateinummer kann optional gesetzt werden.

Dateinummern können jeder beliebige numerische Ausdruck sein. Der Ausdruck muß einer Zahl zwischen 1 und 15 entsprechen. 15 ist die maximal erlaubte Anzahl von Dateien. Andernfalls tritt ein "Bad File Number"-Fehler auf. Entsprechen Dateinummern Realzahlen, wandelt CLOSE diese in Integerzahlen um.

Wenn Sie nach dem Befehlswort CLOSE keine Dateinummer vorgeben, schließt die Anweisung alle derzeit geöffneten Dateien.

ANMERKUNG: NEW, END, RUN, LOAD, OLD, QUIT und SYSTEM schließen alle geöffneten Dateien automatisch. Die STOP-Anweisung schließt keine Disketten-Dateien.

Beispiele:

Die nachfolgende Anweisung schließt alle geöffneten Disketten- Dateien:

```
Ok 310 CLOSE
```

In diesem Beispiel werden die geöffneten Disketten-Dateien, denen die Dateinummern 3 und 7 zugeordnet wurden, geschlossen:

```
Ok 600 CLOSE #3, #7
```

Syntax: CLOSEW <Fenster-Nummer>

Effekt: Schließt eines der BASIC-Fenster.

Erklärung:

CLOSEW wird verwendet, um eines der vier BASIC-Fenster zu schließen. Dieser Aufruf muß zum Schließen aller Fenster für jedes separat vorgenommen werden. Die Angabe <Fenster-Nummer> lautet folgendermaßen:

0 - Bearbeitungsfenster
1 - Auflistungsfenster
2 - Ausgabefenster
3 - Befehlsfenster

Anmerkung: CLOSEW gibt eine interne Meldung an den BASIC- Interpreter, um dem System eine Kontrolle des jeweiligen Fenster- Status zu ermöglichen. Aus diesem Grund sollten Sie BASIC-Fenster niemals über direkte AES-Aufrufe schließen.

Syntax: COLOR <Textfarbe, Füllfarbe, Linienfarbe, Index, Stil>

Effekt: Legt die Text-, Füll- und Zeichenfarbe, sowie die Füllmuster fest.

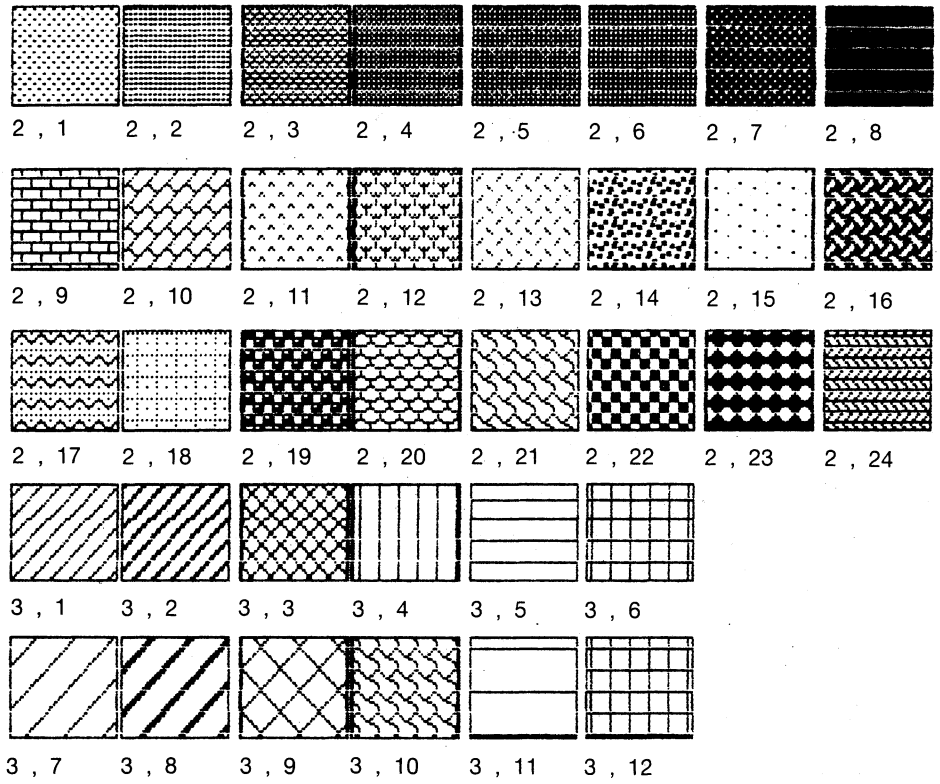
Erklärung:

COLOR setzt die Farbe des im Ausgabefenster gedruckten Textes, des Hintergrundes (Füllfarbe) und der im Ausgabefenster gezeichneten Linien fest. Zudem werden Farben und Muster, mit denen die gezeichneten Formen gefüllt werden sollen, festgelegt. COLOR beeinflußt nachfolgende PRINT und Grafik-Farben, verändert allerdings bereits im Ausgabefenster vorhandene Texte oder Zeichnungen nicht.

Die untenstehende Tabelle zeigt die einzelnen Farbwerte in unterschiedlichen Auflösungen:

FARBWERT	NIEDRIG	MITTEL	HOCH
0	X	X	X
1	X	X	X
2	X	X	
3	X	X	
4	X		
5	X		
6	X		
7	X		
8	X		
9	X		
10	X		
11	X		
12	X		
13	X		
14	X		
15	X		

Die nachfolgende Tabelle zeigt die Muster, die über Parameter 4 und 5 ausgewählt werden können, sowie den verfügbaren Füll-Stil. Unter jedem Kästchen sind zwei Zahlen angegeben, die durch ein Komma voneinander abgetrennt sind. Die Zahl links neben dem Komma gibt den Füll-Stil an (Unausgefüllt, ausgefüllt mit einem Muster, oder ausgefüllt mit einer Schraffierung). Die Zahl rechts neben dem Komma steht für den Index des gewählten Musters bzw. der Schraffierung.



Beispiele:

```
Ok 10 COLOR 1,0,1
Ok 20 PRINT "SCHWARZ"
Ok 30 COLOR 2,0,1
Ok 40 PRINT "ROT"
Ok 50 COLOR 1,0,1
Ok RUN
SCHWARZ        ***** in schwarzer Schrift *****
ROT            ***** in roter Schrift *****
Ok
```

```
Ok 10 COLOR 1,2,3,1,1
Ok 20 FULLW 2: CLEARW 2
Ok 30 K=(K+10) MOD 3600
Ok 40 FOR I=3 TO 11
Ok COLOR 1,1,1,1,2
Ok 60 J=I*400
Ok 70 PCIRCLE 150,80,80,(J+K+3600) MOD 3600, (J+K+400) MOD 3600
Ok 80 NEXT
Ok 90 GOTO 30
```

Syntax: COMMON <Variable> <,Variable>

Effekt: Gibt die Variablen an, die ein Programm an ein mit CHAIN eingebundenes Programm übergeben kann.

Erklärung:

ST BASIC behandelt alle COMMON-Anweisungen in einem Programm als eine einzige konsekutive Liste von Variablen. Aus diesem Grund kann ein Programm eine beliebige Anzahl von COMMON-Anweisungen enthalten.

COMMON-Anweisungen können an jeder beliebigen Stelle innerhalb eines Programmes erscheinen. Am günstigsten ist eine Plazierung dieser Anweisungen am Anfang des Programmes.

COMMON wird in Verbindung mit CHAIN verwendet.

Lesen Sie hierzu auch unter CHAIN nach.

Beispiel:

Das nachfolgende Beispiel bindet in ein Programm mit dem Namen "STBASIC" ein und übergibt die Variablen VAL!, NAME und die Feld-Variable SCALE().

```
Ok  0 COMMON VAL!, NAME$, SCALE()
Ok  0 CHAIN "STBASIC"
```

Syntax: CONT

Effekt: Nimmt einen durch BREAK, STOP oder CTRL-G unterbrochenen Programmlauf wieder auf.

Erklärung:

Eine BREAK- oder STOP-Anweisung in einem Programm, bzw. das Betätigen von CTRL-G (sofern nicht getrapped) bringt ST BASIC in den BREAK-Modus. Im BREAK-Modus können Sie Direktmodus- Anweisungen verwenden, um dazwischenliegende Programmwerte zu verändern.

Mit CONT wird der Programmlauf fortgesetzt.

Sie können auch eine Direktmodus GOTO-Anweisung verwenden, um die Programmausführung in eine spezielle Programmzeile zu leiten.

Beispiel:

```
Ok 10 N = 5
Ok 20 FOR X = 1 TO 5
Ok 30 N = N - 1
Ok 40 PRINT N
Ok 50 NEXT X
Ok RUN
  4
  3
  2
    [drücke [CTRL] [G]]
  -- Break -- at line 30
Ok CONT
  1
  0
Ok
```

Syntax: X = COS (<numerischer Ausdruck>)

Effekt: Gibt den Cosinus einer Zahl aus.

Erklärung:

Die COS-Funktion gibt eine Realzahl mit einfacher Genauigkeit aus. Die Zahl ist der Cosinus-Wert des Winkels im numerischen Ausdruck.

Alle trigonometrischen Funktionen in ATARI BASIC erfordern die Angabe des Winkels in Radian.

Beispiel:

```
Ok 10 PI = 3.14159
Ok 20 DEGREES = 180
Ok 30 RADIANS = DEGREES * (PI/180)
Ok 40 ANS! = COS(RADIANS)
Ok 50 PRINT "der Cosinus ist"; ANS!
Ok RUN
Der Cosinus ist -1
Ok
```

Syntax: CVD(<8-Byte String>)
CVI(<2-Byte String>)
CVS(<4-Byte String>)

Effekt: CVD-, CVI- und CVS-Funktionen wandeln Byte Strings in numerische Variablen um. Dient zur Umwandlung in ASCII-Zahlen, die aus Random-Dateien ausgelesen werden können.

Erklärung:

ST BASIC speichert Zahlen in einer Random-Datei als Byte Strings. Um die Zahlen aus der Datei auszulesen, müssen die Strings in die dazugehörigen numerischen Daten-Typen konvertiert werden. Die Funktionen verändern nicht den Wert der Zahl, sondern lediglich den Daten-Typ. Diese Strings sind die exakte Byte-Repräsentation der gespeicherten Zahlen. Es handelt sich dabei NICHT um Zeichen-Strings, die gedruckt werden können.

Die Funktion CVD konvertiert einen 8-Byte String in eine Realzahl mit doppelter Genauigkeit.

Die Funktion CVI wandelt einen 2-Byte String in eine Integerzahl um.

Die Funktion CVS konvertiert einen 4-Byte String in eine Realzahl mit einfacher Genauigkeit.

Falls der String, der aus der Datei ausgelesen wird, kürzer ist als für die Konvertierung erforderlich, wird er nach rechts mit Binär-Nullen aufgefüllt.

MKD$, MKI$ und MKS$ sind die gegensätzlichen Funktionen zu CVD, CVI und CVS.

Beispiel:

```
Ok 10 OPEN "R",#1,"NUMBERS"
Ok 20 FIELD #1, 2 AS A$, 4 AS B$, 8 AS C$
Ok 30 GET #1, REC%
Ok 40 I% = CVI(A$)
Ok 50 X! = CVS(B$)
Ok 60 Y# = CVD(C$)
Ok 70 PRINT I%, X!, Y#
Ok 80 CLOSE #1
Ok 90 END
```

Über diesen Programmlauf wird ein Satz Zahlen aus der Datei entnommen und ausgedruckt.

Syntax: DATA <Konstante> ,<Konstante>

Effekt: Definiert eine Liste von Konstanten, die eine READ- Anweisung Variablen zuordnen kann.

Erklärung:

DATA-Anweisungen ermöglichen Ihnen, feste Werte Variablen zuzuordnen. Die Werte werden gemäß ihrer Reihenfolge in einer DATA-Anweisung zugeordnet.

Jede DATA-Konstante muß eine dazugehörige READ-Variable besitzen und umgekehrt. Die Konstanten und Variablen passen gemäß der Reihenfolge, in der sie aufgelistet sind, zusammen; die erste DATA-Konstante ist von der ersten READ-Variablen abhängig usw.

DATA-Konstante können Integerzahlen, Realzahlen oder Strings in jeder beliebigen Kombination sein. Die Daten-Typen für die Konstanten in der DATA-Liste müssen jedoch mit den Variablen zusammenpassen, die ihnen in einer READ-Anweisung zugeordnet wurden. Setzen Sie keine Anführungszeichen um Strings in einer DATA-Anweisung.

DATA-Anweisungen können beliebig lang sein. Sie können jedoch in eine Zeile, die eine DATA-Anweisung enthält, keine anderen, zusätzlichen Anweisungen mehr schreiben.

Obwohl jede Konstante eine zugehörige Variable besitzen muß, brauchen Sie nicht für jede DATA nweisung eine READ-Anweisung. Sie können mehrere DATA-Anweisungen ir in Programm einarbeiten, denen Sie innerhalb einer einzigen READ-Anweisung ' riablen zuordnen. In diesem Fall passen sie zuerst gemäß der Reihenfolge der Kc tanten im Programm, und danach gemäß ihrer Reihenfolge innerhalb der Zeilen zi mmen.

Die RESTO E-Anweisung ordnet READ-Anweisungen DATA-Anweisungszeilen zu.

Lesen Sie hierzu auch unter READ und RESTORE nach.

Beispiel:

```
Ok 10 READ X
Ok 20 DATA 33.3, 5, "PLATZRESERVIERUNG"
Ok 30 PRINT X
Ok 40 READ X, Y$
Ok 50 PRINT X,Y$
Ok RUN
  33.3
  5 PLATZRESERVIERUNG
```

Syntax: DEF FN<Funktionsname>[(<Parameter,Parameter>)] = <Definition>

Effekt: Definiert anwenderspezifische Funktionen.

Erklärung:

DEF FN erlaubt Ihnen die Definition eigener Funktionen für die Verwendung in einem Programm. Der Name für die Funktion kann ein beliebiger, gültiger Variablenname sein.

Die Variablenliste in Klammern ist optional. Sie können jeden Variablen-Typ mit Ausnahme von Arrays verwenden. Diese Variablen sind an die definierte Funktion gebunden und beeinflussen Variablen mit demselben Namen an einer anderen Stelle im Programm nicht. Die Variablen in Klammern können als "Platzreservierungen" für die Werte, die beim Aufrufen der Funktion an sie übertragen werden, angesehen werden. Die Werte, die Sie an die Funktion übertragen, müssen bezüglich des Typs und der Zahl mit den Werten in Klammern zusammenpassen.

Sie können beliebige globale Variablen innerhalb der Funktionsdefinition in Ihrem Programm verwenden. Sie werden genauso behandelt wie von der Funktionsdefinition angegeben. Wenn Sie ihre Werte innerhalb der Funktion verändern, behalten sie ihre neuen Werte während des gesamten Programmes bei.

Die Definition ist ein Ausdruck, der die Arbeitsweise der Funktion festlegt. Die Beschreibung ist auf eine Programmzeile beschränkt. Enthält der Funktionsname eine Typen-Spezifikation wie beispielsweise FNA$, muß die Definition mit diesem Typ zusammenpassen. Die Parameter, die an die Funktion übertragen werden (in Klammern gesetzt) müssen sich ebenfalls an diesen Typ anpassen.

Beispiel:

```
Ok 10 INPUT "BREITE MATERIAL IN CM";
MATERIAL.BREITE
Ok 20 INPUT "BREITE FENSTERBANK IN CM";
FENSTER.BREITE
Ok 30 BRETTER.NOETIG = FENSTER.BREITE / MATERIAL.BREITE
Ok 40 INPUT "LAENGE FENSTERBANK IN CM";
FENSTER.LAENGE
Ok 50 MENGENMETER.NOETIG = BRETTER.NOETIG
* FENSTER.LAENGE
Ok 60 INPUT "MATERIALPREIS PRO METER";
PREIS.METER!
Ok 70 DEF FNMATERIAL = METER.NOETIG /
15 + MENGENMETER.NOETIG
Ok 80 DEF FNKOST! = (PREIS.METER) * FNMATERIAL
Ok 90 PRINT "SIE BRAUCHEN ";MATERIAL;" CM AN
";MATERIAL.BREITE;" CM MATERIAL.":PRINT "IHRE KOSTEN
SIND: ";FNKOST!
Ok 100 DEF FNIN METERN = MATERIAL / 36
```

```
Ok 110 PRINT MATERIAL; CM IN METERN IST "; FNINMETERN
Ok RUN
BREITE MATERIAL IN CM? 30
BREITE FENSTERBANK IN CM? 60
LAENGE FENSTERBANK IN CM? 60
MATERIALPREIS PRO METER? 2.00
SIE BRAUCHEN 128 CM AN 30 CM MATERIAL. IHRE KOSTEN
SIND 7.1111
 128 CM IN METER IST 3.5556
Ok
```

Syntax: DEF SEG [<numerischer Ausdruck>]

Effekt: DEF SEG etabliert den Operationsmodus von PEEK und POKE, sowie das von den Befehlen verwendete Offset.

Erklärung:

Die Operationsmodi werden wie folgt definiert:

Ist DEF SEG > 0, wird ein Byte gePEEKt oder gePOKEt. Der in DEF SEG verwendete Wert des numerischen Ausdruckes wird als Offset für die in PEEK oder POKE spezifizierte Adresse angesehen.

Ist DEF SEG = 0, werden zwei Bytes gePEEKt oder gePOKEt. Der Wert des in DEF SEG verwendeten numerischen Ausdruckes wird als Offset für die in PEEK oder POKE spezifizierte Adresse angesehen.

Wenn DEF SEG = 0 ist und gleichzeitig die Adresse durch DEFDBL angegeben wurde, werden vier Bytes (Long-Integer) gePEEKt oder gePOKEt.

Beispiele:

```
82.6 Ok 10 DEF SEG=0
Ok 20 DEFDBL S:S=SYSTAB+20:'ZEIGER GRAFIK-SPEICHER
Ok 30 X=PEEK(S):'xIST EIN 4-BYTE WERT
Ok 40 RESET:'LEGT DEN AKTUELLEN BILDSCHIRM IM GRAFIK-
   SPEICHER AB
Ok 50 BSAVE "BILD",X,32767
Ok 60 CLEARW 2: 'LOESCHE BILDSCHIRM
Ok 70 BLOAD "BILD",X: LADEN DES BILDES IN DEN GRAFIK-
   SPEICHER
Ok 80 OPENW 2: 'DARSTELLUNG DES GRAFIK-SPEICHERS
   IM FENSTER

Ok 10 DEF SEG=100
Ok 20 PRINT PEEK(500)
```

Anmerkung: Damit wird eine 1-Byte Integerzahl aus der absoluten Speicherstelle 600 ausgegeben.

```
Ok 10 DEF SEG=0
Ok 20 LOC#=175000
Ok 30 PRINT PEEK(LOC#)
```

Anmerkung: Damit wird eine 4-Byte Integerzahl aus der Speicherstelle 175000 ausgegeben.

Syntax: DEFDBL <Buchstabe> <-Buchstabe>

Effekt: Gibt einen Buchstaben, bzw. einen Bereich von Buchstaben an, der als Realzahl mit doppelter Genauigkeit definiert werden soll.

Erklärung:

Die DEFDBL-Anweisung gibt vor, daß die Variablen, deren Namen mit einem der angegebenen Buchstaben beginnen, Realzahlen mit doppelter Genauigkeit sind. Sie können einen einzelnen Buchstaben oder einen Bereich von Buchstaben, wie beispielsweise A-D, als Parameter angeben.

Typenangabe-Zeichen haben generell Vorrang vor DEFDBL-Anweisungen. DEFDBL-Anweisungen können nur als erste Anweisungen in einem Programm eingegeben werden. DEFDBL wird immer in Verbindung mit DEF SEG, PEEK oder POKE verwendet.

Anmerkung: DEFDBL-Anweisungen verändern die ST BASIC Interpretation von Programmzeilen.

Beispiel:

```
Ok 10 DEFDBL X-Y
Ok 20 X = 123123412345123456
Ok 30 Y = $H333
Ok 40 PRINT X,Y
Ok RUN
  1.23123392D+017          819
Ok
```

Syntax: DEFINT <Buchstabe> <-Buchstabe>

Effekt: Gibt einen Buchstaben oder Bereich von Buchstaben an, der als Integerzahl definiert werden soll.

Erklärung:

Die DEFINT-Anweisung gibt vor, daß die Variablen, deren Namen mit einem der angegebenen Buchstaben beginnen, Integerzahlen sind. Sie können einen einzelnen Buchstaben oder einen Bereich von Buchstaben, wie beispielsweise M-Z, als Parameter verwenden.

Typenangabe-Zeichen haben generell Vorrang vor DEFINT-Anweisungen.

Anmerkung:DEFINT-Anweisungen verändern die ST BASIC Interpretation von Programmzeilen. Wenn Sie eine Variable mit der DEFINT-Anweisung als Integerzahl vorgeben, behandelt ST BASIC sie auch dann als Integerzahl, wenn Sie die DEFINT-Anweisung nachträglich löschen.

Beispiel:

```
Ok 10 DEFINT X-Y
Ok 20 X = 78.9
Ok 30 Y = 78.1
Ok 40 PRINT X,Y
Ok RUN
  78          78
Ok
```

Syntax: DEFSNG <Buchstabe> <-Buchstabe>

Effekt: Gibt einen Buchstaben oder Bereich von Buchstaben an, der als Realzahl definiert werden soll.

Erklärung:

Die DEFSNG-Anweisung definiert die Variablennamen, die mit einem der angegebenen Buchstaben beginnen, als Realzahlen. Sie können einen einzelnen Buchstaben oder einen Bereich von Buchstaben, wie beispielsweise A-D, als Parameter verwenden.

Typenangabe-Zeichen haben generell Vorrang vor DEFSNG- Anweisungen.

Anmerkung: DEFSNG-Anweisungen verändern die ST BASIC Interpretation von Programmzeilen.

Beispiel:

```
Ok 10 DEFSNG X-Y
Ok 20 X = 23D+16
Ok 30 Y = 456654456654
Ok 40 PRINT X,Y
Ok RUN
  2.3E+17          4.56654E+11
```

Syntax: DEFSTR <Buchstabe> <-Buchstabe>

Effekt: Gibt einen Buchstaben oder Bereich von Buchstaben an, der als Zeichenkette definiert werden soll.

Erklärung:

Die DEFSTR-Anweisung gibt vor, daß alle Variablen, deren Anfangsbuchstaben in der Parameterliste aufgeführt sind, Strings sind. Sie können einen einzelnen Buchstaben oder einen Bereich von Buchstaben, wie beispielsweise M-Z, als Parameter verwenden.

Typenangabe-Zeichen haben generell Vorrang vor DEFSTR- Anweisungen. Der vorgegebene Variablentyp ist eine Realzahl.

Anmerkung: DEFSTR-Anweisungen verändern die ST BASIC Interpretation von Programmzeilen. Wenn Sie eine Variable mit der DEFSTR-Anweisung als String vorgeben, behandelt ST BASIC sie auch dann als Realzahl, wenn Sie die DEFSTR-Anweisung nachträglich löschen.

Beispiel:

```
Ok 10 DEFSTR A-C
Ok 20 A = "12.7.42"
Ok 30 B = "1066"
Ok 40 C = "4.12.XX"
Ok 50 PRINT A,B,C
Ok RUN
12.7.42         1066         4.12.XX
Ok
```

DELETE - 40
DELETE 20
DELETE 20, 30
DELETE 20 - 30

Syntax: DELETE <Zeilennummer Liste>

Effekt: DELETE löscht Programmzeilen aus dem Arbeitsspeicher.

Erklärung:

DELETE löscht die von Ihnen angegebenen Programmzeilen. Das Löschen einer einzelnen Programmzeile erfolgt besser über die Eingabe der Zeilennummer und Betätigen der RETURN-Taste.

Beispiel:

```
Ok 10 X = 10
Ok 20 Z = 10
Ok 30 PRINT X,Z
Ok DELETE 20-30
Ok LIST
Ok 10 X = 10
Ok
```

DIM **DIM A$ (5)** **ANWEISUNG**
DIM X (5, 10, 4)
DIM B$(10) ,C$(20)
DIM X(5,10,4) ,Y(1,2,8)

Syntax: DIM<Array-Name>(<Unterbereich> <,Unterbereich>)
(,<Array-Name>[<Unterbereich>])

Effekt: Definiert die Anzahl der Dimensionierungen und die Anzahl der Elemente in einem Array.

Erklärung:

Die DIM-Anweisung reserviert Platz für einen String oder ein numerisches Array durch Spezifizieren der Anzahl von Dimensionierungen und der oberen Grenze der Elemente in jeder Dimension. Die Anzahl der Dimensionierungen ist abhängig von der Anzahl der Unterbereiche. Ein Unterbereich entspricht einer Dimensionierung, zwei Unterbereiche entsprechen zwei Dimensionierungen usw. Die Anzahl der Elemente und Dimensionierungen, die Sie spezifizieren können, ist abhängig vom verfügbaren Speicherplatz. Allerdings ist die maximale Anzahl der Dimensionierungen in jedem Fall auf 15 begrenzt.

Die untere Grenze jeder Dimensionierung ist 0 oder 1, abhängig von OPTION BASE.

DIM setzt den Anfangswert der Elemente automatisch auf 0.

In ST BASIC sind Arrays dynamisch. Sie können Arrays mit DIM dimensionieren, das Array später im Programm löschen und dann mit DIM unter Verwendung desselben Namens, aber mit neuen Dimensionierungen erneut einrichten. Bei dynamischen Arrays können Sie das Array auch unter Verwendung einer numerischen Variablen dimensionieren.

Sie können ein Array verwenden, ohne es zuvor mit einer DIM- Anweisung zu definieren. In diesem Fall wird für das Array automatisch eine obere Grenze von 10 Elementen in jeder Dimension vorgegeben. Ist beispielsweise die erste Referenz zu ARRAY A

```
ARRAY A(7,3)
```

wird das Array so eingerichtet, als wäre es mit

```
DIM A(10,10)
```

definiert worden.

Die vorgegebene Anzahl von erlaubten Dimensionen ist 4 für Integerzahlen und 3 für Strings sowie Realzahlen.

Anmerkung: In ST BASIC können 30 % des verfügbaren Speicherplatzes als Arrays bezeichnet werden. Allerdings dürfen alle Arrays zusammen nicht mehr als 32 K umfassen, unabhängig vom insgesamt verfügbaren Speicherplatz.

Beispiel:

```
Ok 10 DIM HAEUSER$ (1,1,1)
Ok 20 HAEUSER$ (0,0,0) = "ETAGENPLAN1"
Ok 30 HAEUSER$ (0,0,1) = "ETAGENPLAN3"
Ok 40 HAEUSER$ (0,1,0) = "ETAGENPLAN3"
Ok 50 HAEUSER$ (0,1,1) = "ETAGENPLAN3"
Ok 60 HAEUSER$ (1,0,0) = "ETAGENPLAN1"
Ok 70 HAEUSER$ (1,0,1) = "ETAGENPLAN2"
Ok 80 HAEUSER$ (1,1,1) = "ETAGENPLAN2"
Ok 90 IF HAEUSER$ (1,0,0) = "ETAGENPLAN2" THEN GOTO 300
```

DIR
DIR A:
DIR B:BAS.PRG
DIR B:*.PRG
DIR B:BAS.*
DIR B:*.*
DIR B:BAS.PR?

Syntax: DIR [<Laufwerk:>][<Dateiname, Dateiart>]

Effekt: Listet die Dateien einer Diskette auf.

Erklärung:

Der Befehl DIR zeigt das Verzeichnis der in das angesteuerte Laufwerk eingelegten Diskette an.

Sie können angeben, welches Laufwerk und welche Dateien Sie anzeigen lassen wollen. Das Sternchen [*] und das Fragezeichen [?] gelten als "Joker"-Benennungen.

Das Zeichen [*] zeigt eine "Unbedeutend"-Spezifikation für ein beliebiges Feld an. So bedeutet *.BAS, daß jede Datei mit dem Extender .BAS aufgelistet werden soll. FIG.* gilt entsprechend für alle Dateiarten mit dem Namen FIG, und B*.BAS für alle Dateiarten mit dem Extender .BAS, die mit dem Anfangsbuchstaben B beginnen.

Das Fragezeichen [?] gilt als "Unbedeutend"-Spezifikation für ein einzelnes Zeichen. So bedeutet ?IG.BAS, daß jede Datei, deren Name drei Buchstaben umfaßt und mit IG.BAS endet (z.B. BIG.BAS, PIG.BAS, FIG.BAS usw.), aufgelistet wird.

Beispiel:

Ok	DIR	Verzeichnis aller Dateien auf der eingelegten Diskette
Ok	DIR A:	Verzeichnis aller Dateien auf Diskette A
Ok	DIR B:BAS.PRG	Sucht nach der Datei BAS.PRG auf Diskette B
Ok	DIR B:*.PRG	Verzeichnis aller Dateien mit dem Extender .PRG auf Diskette B
Ok	DIR B:BAS.*	Verzeichnis aller Dateien jeder Art mit dem Namen BAS auf Diskette B
Ok	DIR B:*.*	Verzeichnis aller Dateien jeder Art auf Diskette B
Ok	DIR B:BAS.PR?	Verzeichnis aller Dateien auf Diskette B, die mit BAS beginnen und über einen Extender mit den Anfangsbuchstaben PR verfügen.

Syntax: EDIT <Zeilennummer> ED <Zeilennummer>

Effekt: Ruft den ST BASIC Editor auf.

Erklärung:

Der Befehl EDIT ruft den ST BASIC Editor auf. Sie können eine Zeilennummer angeben, in der mit dem Editieren begonnen werden soll. Wird keine Zeilennummer vorgegeben, beginnt EDIT in der ersten Zeile des derzeit im Speicher befindlichen Programmes.

Syntax: ELLIPSE <horizont. Mittelpunkt, vertik. Mittelpunkt, horizont. Radius, vertik. Radius>[<,Anfangswinkel, Endwinkel>]

Effekt: ELLIPSE zeichnet Ellipsen und Kreisausschnitte.

Erklärung:

ELLIPSE zeichnet eine Ellipse, deren Mittelpunkt an dem Punkt liegt, der durch die ersten beiden Parameter (horizontaler und vertikaler Mittelpunkt) vorgegeben wurde. Die Positionen werden in Pixel angegeben und von der oberen linken Ecke des Ausgabefensters aus gezählt.

Der dritte und vierte Parameter (horizontaler und vertikaler Radius) wird ebenfalls in Pixel ausgedrückt. Die horizontale und vertikale Pixelangabe ist abhängig von der gewählten Auflösung, sowie von der Größe des Ausgabefensters.

Die Ellipse wird in der festgelegten Zeichenfarbe (Parameter 3 der COLOR-Anweisung) dargestellt.

Die letzten beiden Parameter (Anfangs- und Endwinkel) sind optional. Wird hier nichts angegeben, zeichnet ELLIPSE eine vollständige Ellipse. Andernfalls zieht ELLIPSE einen Ellipsenausschnitt zwischen den beiden Punkten. ELLIPSE zeichnet allerdings nur einen Kreisbogen und kein eingefärbtes Kreissegment. Winkel werden in Grad mal 10 ausgedrückt. So werden 45 Grad als 450, 180 Grad als 1800 usw. angegeben. O Grad zeigt zum rechten Rand des Fensters, 90 Grad zum oberen, 180 Grad zum linken und 270 Grad zum unteren Fensterrand. ELLIPSE 100,80,40,50,0,3600 zeichnet eine vollständige Ellipse.

Lesen Sie auch unter PELLIPSE, CIRCLE und PCIRCLE nach.

Beispiel:

```
Ok 10 COLOR 1,0,1:CLEARW 2
Ok 20 ELLIPSE 100,80,40,80
Ok 30 COLOR 1,0,2
Ok 40 ELLIPSE 100,80,40,80,300,90
Ok RUN
[Im Ausgabefenster wird eine schwarze Ellipse mit einem roten Kreisbogen über
60 Grad, beginnend bei 30 Grad, gezeichnet]
Ok
```

Syntax: END

Effekt: Beendet einen Programmlauf, schließt alle Dateien und kehrt zum Befehls-Level zurück.

Erklärung:

Sie können eine END-Anweisung an jede beliebige Stelle, an der Sie zum Befehls-Level zurückkehren wollen, setzen. Ein END am Ende des Programmes kann optional verwendet werden.

END unterscheidet sich von STOP dahingehend, daß hier alle Dateien geschlossen werden und zum Befehls-Level zurückgekehrt, jedoch keine STOP-Meldung ausgegeben wird.

Beispiel:

```
Ok 10 PRINT "DAS PROGRAMM"
Ok 20 PRINT "LAEUFT"
Ok 30 PRINT "ABER ES ERREICHT"
Ok 40 PRINT "NIEMALS DAS LETZTE"
Ok 50 PRINT "WORT DIESES"
Ok 60 END
Ok 70 PRINT "PROGRAMMES"
Ok RUN
DAS PROGRAMM
LAEUFT
ABER ES ERREICHT
NIEMALS DAS LETZTE
WORT DIESES
Ok
```

Syntax: X = EOF (<Dateinummer>)

Effekt: Gibt TRUE (-1) am Ende einer sequentiellen oder Random Access - Datei aus.

Erklärung:

Wenn Sie in eine sequentielle Datei schreiben, wird automatisch dessen Ende markiert. Versuchen Sie, über das Ende der Datei hinaus zu lesen, tritt ein Fehler auf. Mit EOF können Sie überprüfen, ob Sie sich am Dateiende befinden.

EOF gibt -1 aus, wenn Sie sich am Dateiende befinden. Andernfalls wird 0 ausgegeben.

Beispiel:

```
Ok 100 INPUT "DATEI ";F$
Ok 110 IF LEN(F$) = 0 THEN END
Ok 120 ON ERROR GOTO 20000
Ok 130 OPEN "I",1,F$
Ok 140 WHILE NOT EOF(1)
Ok 150 LINE INPUT #1,R$: ?R$
Ok 160 WEND
Ok 200 ?:CLOSE 1: GOTO 100
Ok 20000 IF ERR = 53 THEN ?"DATEI";F$;
" NICHT GEFUNDEN": RESUME 100 ELSE ON ERROR GOTO 100
```

ERA **ERA DATEI.TXT** **BEFEHL**
ERA B: DATEI.TXT

Syntax: ERA [<Laufwerk:>]<Dateiname>

Effekt: Löscht eine Datei von Diskette.

Erklärung:

Der Befehl ERA löscht alle Dateien mit dem angegebenen Dateinamen von der Diskette im angesteuerten Laufwerk. Eine gelöschte Datei kann nicht wieder zurückgeholt werden.

ERASE **ERASE A$, B$, C** **ANWEISUNG**

Syntax: ERASE <Array-Name> ,<Array-Name>

Effekt: Löscht Arrays.

Erklärung:

ERASE löscht ein Array, so daß Sie es neu dimensionieren oder den dadurch belegten Speicherplatz wiedergewinnen können. Sie müssen Felder grundsätzlich löschen, bevor Sie sie neu dimensionieren.

Lesen Sie hierzu auch unter DIM nach.

Beispiel:

```
Ok 10 DIM RECHNUNG$ (10,10)
Ok 20 RECHNUNG$ (0,0) = "BEYELSTEIN, KARIN"
Ok 30 ERASE RECHNUNG$
Ok 40 DIM RECHNUNG$(5,5,5)
```

Syntax: X = ERL
X = ERR

Effekt: Die Variablen ERL und ERR sind reservierte Variablen, die in Unterroutinen für Fehlerbehandlung verwendet werden.

Erklärung:

ERL enthält die Zeilennummer, in der ein Fehler auftrat. ERR enthält den Fehler-Code. ERL und ERR sind reservierte Variablen, was bedeutet, daß Sie diese nicht links neben ein Gleichzeichen in einer Zuordnungs-Anweisung schreiben dürfen.

Ist die Anweisung oder der Befehl, in dem der Fehler auftrat, im Direkt-Modus geschrieben, ist der Wert von ERL Null. Tritt im Direkt-Modus ein Fehler auf, wird der Programmlauf grundsätzlich angehalten.

Ist die Anweisung im indirekten Modus geschrieben, schreiben Sie IF-Anweisungen wie folgt:

IF ERL = <Fehlerzeile> THEN <auszuführende Anweisung>
IF ERR = <Fehler-Code> THEN <auszuführende Anweisung>

Lesen Sie hierzu auch unter ERROR nach, um weitere Informationen über Fehleraufdeckung, sowie Beispiele für die Verwendung von ERL und ERR in Fehlerbehandlungs-Unterroutinen zu erhalten.

Syntax: ERROR<numerischer Ausdruck>

Effekt: Simuliert einen BASIC Laufzeitfehler und übergibt die Kontrolle an eine Fehleraufdeckungs-Routine.

Erklärung:

Sie können Fehler und Fehlermeldungen in Ihren Programmen mit der ERROR-Anweisung definieren. ERROR ordnet einem Fehler eine Fehler-Codezahl zu. Die Zahl muß ein Integer-Ausdruck sein.

Bei jedem Auftreten des Fehlers bezieht sich das Programm auf die Fehler-Codezahl. Entspricht der Fehler-Code einem ST BASIC Fehler-Code, wird die ST BASIC Fehlermeldung ausgedruckt. Ist dagegen eine von Ihnen geschriebene Fehleraufdeckung aktiviert, wird die Programmkontrolle an Ihre Fehleraufdeckungs-Routine übergeben.

Zwei vordefinierte Variablen sind der ERROR-Anweisung zugeordnet: ERL und ERR.

Beim Auftreten eines Fehlers enthält ERR die Fehlercode-Konstante. Diese kann zum Schreiben von Fehlermeldungen verwendet werden (z.B.:
IF ERR = 100 THEN PRINT "BITTE ZAHL PRUEFEN UND NEU EINGEBEN".)

ERL enthält die Zeilennummer, in der der Fehler auftrat.

Wurde keine Anwender-Fehleraufdeckung gesetzt, wird die Meldung, die dem Wert in ERR entspricht, ausgegeben. Das Programm wird angehalten. Dieser Fall tritt auch ein, wenn eine ERROR-Anweisung im Direkt-Modus ausgeführt wird, unabhängig davon, ob Sie eine Fehleraufdeckung gesetzt haben oder nicht.

Wenn Sie eine Fehleraufdeckung setzen, springt das Programm in die Fehleraufdekkungs-Routine. Sie können ERR und ERL wie jede beliebige numerische Variable verwenden. Um die Fehleraufdeckung zu verlassen, verwenden Sie RESUME. Dabei ist es unerheblich, ob in die Fehleraufdeckung aufgrund eines aufdeckbaren ST BASIC Fehlers, oder aufgrund einer ERROR-Anweisung gesprungen wurde.

Entspricht der Fehler-Code einem vordefinierten ST BASIC Fehler-Code, simuliert das Programm den Fehler und druckt die Fehlermeldung für diesen Code. Die ST BASIC Fehlermeldungen finden Sie im Anhang D dieses Handbuches.

Wenn Sie eigene Fehler definieren, sollten Sie möglichst Ihren Fehler-Codes Werte geben, die erheblich größer sind als die Codes von ST BASIC. In diesem Fall wird es niemals erforderlich werden, Ihr Programm nachträglich zu verändern, wenn zu einem späteren Zeitpunkt die Fehler-Codes von ST BASIC überarbeitet und verändert werden sollten.

Lesen Sie hierzu auch unter ON ERROR GOTO, GOTO und RESUME nach.

Beispiele:

Sie können Fehler sowohl im direkten, als auch im indirekten Modus simulieren. Nachfolgend sehen Sie ein Beispiel für Direkt-Modus:

```
Ok ERROR 55
Eine bereits geöffnete Datei kann nicht gelöscht (KILL) oder geöffnet (OPEN)
werden
```

Nachfolgendes Beispiel betrifft den Indirekt-Modus:

```
.
.
.
Ok 500 ON ERROR GOTO 550
Ok 510 INPUT "WOLLEN SIE EINEN DISPOSITIONSRAHMEN FUER
   IHR KONTO"; E$
Ok 515 IF E$ = "NEIN" THEN GOTO 600
Ok 520 INPUT " IST DER IN ZEILE 33 AUFGEFUEHRTE BETRAG
   KLEINER ALS $10,000";X$
Ok 525 IF X$ = "NEIN" THEN ERROR 200
Ok 530 IF ERR = 200 THEN
Ok 535 PRINT "SIE SIND NICHT KREDITWUERDIG."
Ok 540 IF ERL = 525 THEN GOTO 600
Ok 550 RESUME
.
.
.
Ok RUN
WOLLEN SIE EINEN DISPOSITIONSRAHMEN FUER IHR KONTO? JA
IST DER IN ZEILE 33 AUFGEFUEHRTE BETRAG KLEINER ALS
$10,000? NEIN SIE SIND NICHT KREDITWUERDIG.
```

Syntax: X = EXP (<numerischer Ausdruck>)

Effekt: Gibt die Konstante e, erhöht um einen Exponenten, aus.

Erklärung:

Die Konstante e ist die Basis natürlicher Logarithmen und entspricht ungefähr 2.7182. EXP gibt eine Realzahl aus.

Der numerische Ausdruck muß kleiner gleich 43.6682 sein.

Beispiel:

```
Ok 10 X = EXP(3.254)
Ok 20 Y = EXP(8.97)
Ok 30 PRINT X,Y
Ok RUN
  25.8937 7863.59
Ok
```

Syntax: FIELD #<Dateinummer>,<Feldbreite> AS <String-Variable> <,Feldbreite> AS <String-Variable>

Effekt: Weist Variablen Plätze in zufällig gewählten Datei- Speichern zu.

Erklärung:

Sie müssen eine FIELD-Anweisung schreiben, um Informationen zwischen Random-Dateidisketten und -speichern zu übertragen. Die FIELD-Anweisung ordnet lediglich Variablen-Plätze zu; sie verschiebt keine Daten.

Die Dateinummer ist die Zahl, die Sie der Datei beim Öffnen zugewiesen hatten. Die Feldbreite definiert die Anzahl der Bytes, die an die String-Variable gegeben werden sollen. So ordnet beispielsweise FIELD #10, 20 AS X$, 30 AS Z$ die ersten 20 Bytes Speicher X$, und die nächsten 30 Bytes Z$ zu.

Sie können nicht mehr Speicherplatz zuordnen als Sie beim Öffnen der Datei geschaffen hatten. Die vorgegebene Datensatzlänge ist 128 Bytes. Sie können für jede Datei beliebig viele FIELD-Anweisungen schreiben.

Eine Neuzuweisung von Feldplätzen löscht die ursprüngliche Aufzeichnung (mapping) nicht. Im Gegenteil können zwei Maps zusammen bestehen. Wenn Sie beispielsweise

```
FIELD #10, 20 AS X$, 40 AS Z$, 10 AS Y$
```

und

```
FIELD #10, 70 AS N$
```

spezifizieren, sind die ersten 20 Bytes von N$ ebenfalls in X$, die nächsten 40 Bytes auch in Z$, und die letzten 10 Bytes auch in Y$enthalten.

Verwenden Sie niemals INPUT oder LET, um Eingaben in eine Variable vorzunehmen, die in einer FIELD-Anweisung deklariert wurde. In diesem Fall würde der Variablen-Zeiger an den String-Platz, und nicht in den Speicher bewegt.

Beispiel:

```
Ok 100 OPEN "R", #5, "STEUER", 40
Ok 110 FIELD #5, 20 AS I$, 10 AS D$, 10 AS E$
```

FILL **FILL 150,80** **ANWEISUNG**

Syntax: FILL <numerischer X-Ausdruck>,<numerischer Y- Ausdruck>

Effekt: Füllt Formen mit Farben oder Mustern aus.

Erklärung:

Füllt gezeichnete Formen mit Farben oder Mustern, die zuvor in einer COLOR-Anweisung definiert wurden, aus. Die X- und Y- Koordinaten bezeichnen die Anfangsposition für FILL.

Lesen Sie hierzu auch unter COLOR nach.

Beispiel:

```
Ok 10 COLOR 1,2,1
Ok 20 CIRCLE 150,80,80
Ok 30 FILL 150,80
Ok 40 COLOR 1,1,1,4,4
Ok 50 FILL 150,80
```

FIX **X = FIX(Y)** **FUNKTION**

Syntax: X = FIX(Zahl)

Effekt: Verkürzt eine Realzahl in eine Integerzahl.

Erklärung:

FIX rundet Zahlen nicht auf oder ab, sondern entfernt lediglich die Dezimalstellen hinter dem Komma. Der Integer-Ausdruck muß zwischen -32768 und 32767 liegen.

Lesen Sie hierzu auch unter CINT und INT nach.

Beispiel:

```
Ok 10 X = 239.77
Ok 20 PRINT FIX(X)
Ok 30 PRINT FIX(-678.3)
Ok RUN
  239
 -678
Ok
```

Syntax: X = FLOAT (<Integer-Ausdruck>)

Effekt: Wandelt eine Integerzahl in eine Realzahl um.

Erklärung:

FLOAT verändert die Darstellung der Integerzahl nicht, odnet ihr jedoch mehr Speicherplatz zu. Der Integer-Ausdruck muß zwischen -32768 und 32767 liegen.

Beispiel:

```
Ok 10 X = FLOAT(97)
Ok 20 PRINT X
Ok RUN
  97
```

Syntax: FOLLOW <Variable>[,<Variable>]

Effekt: Verfolgt die Werte von Programm-Variablen.

Erklärung:

Der Befehl FOLLOW ist eine Fehlerbehebungshilfe, die die Übersicht über alle Programm-Variablen behält. Nach jedem Verändern des Wertes einer angegebenen Variablen druckt FOLLOW den Variablennamen, ihren Wert und die Programmzeile, in der die Veränderung eintrat, aus. Der Befehl UNFOLLOW beendet FOLLOW.

Beispiel:

```
Ok 10 FOR X=1 TO 3
Ok 20 N = N + 1
Ok 30 B = B + 1
Ok 40 PRINT N
Ok 50 PRINT B
Ok 60 NEXT X
Ok RUN
  1
  1
  2
  2
  3
  3
Ok FOLLOW N,B
Ok RUN
N! = 1 at line 20
B! = 1 at line 30
  1
  1
N! = 2 at line 20
B! = 2 at line 30
  2
N! = 3 at line 20
B! = 3 at line 30
  3
Ok UNFOLLOW
Ok
```

Syntax: FOR <Zähl-Variable> = <numerischer Ausdruck> TO <numerischer Ausdruck> [STEP<numerischer Ausdruck>]

Effekt: Erstellt eine Schleife, die so oft ausgeführt wird, wie angegeben wurde.

Erklärung:

Die FOR-Anweisung setzt die Anfangs- und Endwerte einer Zähl- Variablen, sowie den in jeder ausgeführten FOR...NEXT Schleife hinzuzuaddierenden Wert fest.

Der Wert, der zu der Zähl-Variablen hinzuaddiert wird, ist grundsätzlich 1, sofern Sie nicht mit STEP eine unterschiedliche Erhöhung vorgeben. Der Wert hinter STEP kann positiv oder negativ sein.

NEXT bewirkt, daß die Instruktionen zwischen FOR und NEXT ausgeführt werden, solange der Wert der Zähl-Variablen kleiner als der durch TO vorgegebene Endwert ist. Ist der absolute Wert der Zähl-Variablen größer als der absolute Endwert, wird die Programmausführung an die hinter NEXT folgende Programmzeile weitergegeben.

Sie können FOR...NEXT Anweisungen auch verschachteln. Das bedeutet, daß Sie innerhalb einer Schleife eine weitere Schleife einbringen können. Wenn Sie Schleifen verschachteln, muß die NEXT-Anweisung für die innere Schleife vor die der äußeren Schleife gesetzt werden.

Lesen Sie hierzu auch unter NEXT nach.

Beispiele:

```
Ok 10 FOR X = 1 TO 5
Ok 20 PRINT X
Ok 30 NEXT
Ok 40 PRINT "DER WERT DER ZAEHL-VARIABLEN IST "X
Ok RUN
  1
  2
  3
  4
  5
DER WERT DER ZAEHL-VARIABLEN IST 6
Ok
```

```
Ok 10 FOR X = 2 TO 1 STEP -1
Ok 20 FOR Y = 1 TO 5
Ok 30 PRINT X Ok 40 PRINT Y
Ok 50 NEXT Y
Ok 60 NEXT X
Ok RUN
Ok
  2
  1
  2
  2
  2
  3
  2
  4
  2
  5
  1
  1
  1
  2
  1
  3
  1
  4
  1
  5
Ok
```

FRE **X = FRE(0)** **FUNKTION**

Syntax: X = FRE (<Test-Argument>)

Effekt: Gibt die Anzahl nicht verwendeter Bytes im Arbeitsspeicher aus.

Erklärung:

FRE erfordert ein Test-Argument. Verwenden Sie ein beliebiges Argument, um die Anzahl freier Bytes im aktuellen Speichersegment zu erfahren.

Beispiel:

```
Ok PRINT FRE(0)
  43000
```

Anmerkung: Der Umfang der BASIC-Arrays ist auf 32 K beschränkt, unabhängig vom verfügbaren Speicherplatz. Die Arrays dürfen nicht mehr als ein Drittel des gesamten zur Verfügung stehenden Speicherplatzes einnehmen.

FULLW **FULLW 2** **ANWEISUNG**

Syntax: FULLW <numerischer Ausdruck>

Effekt: Erweitert BASIC-Fenster auf die volle Bildschirmgröße.

Erklärung:

FULLW vergrößert das angegebene Fenster auf den vollen Bildschirm-Umfang. Die Fenster werden wie folgt angegeben:

0 = Bearbeitungsfenster
1 = Auflistungsfenster
2 = Ausgabefenster
3 = Befehlsfenster

Beispiel:

```
Ok 10 FULLW 2:CLEARW 2
Ok 20 PRINT "HALLO"
Ok RUN
```

Syntax: GET [#]<Dateinummer> [,<Datensatz-Nummer>]

Effekt: Liest einen Datensatz von einer Random- Diskettendatei in den Datei-Speicher.

Erklärung:

Die Dateinummer ist die Zahl, die Sie der Datei beim Öffnen zugewiesen hatten. Die Datensatz-Nummer ist optional. Falls Sie diese Angabe entfallen lassen, wird der Datensatz, der der ersten GET- oder PUT-Angabe folgt, in den Speicher eingelesen. Die größte verwendbare Datensatz-Nummer ist 32767.

Lesen Sie unter OPEN nach, um ein Beispiel für die Verwendung von GET im Zusammenhang zu erhalten.

Beispiel:

```
Ok 100 IF X$ = "JA" THEN GET#5, TYPE%: GOTO 200
```

Syntax: GOSUB <Zeilennummer> oder GOSUB <Sprungmarken-Name>

Effekt: Gibt die Programmkontrolle an eine Unterroutine ab.

Erklärung:

Die GOSUB-Anweisung ist mit der RETURN-Anweisung kombiniert, die die Programmkontrolle an die direkt nach der GOSUB-Anweisung folgende Programmanweisung zurückgibt.

Die Zeilennummer oder Sprungmarke zeigt die Zeile an, in der die Unterroutine beginnt.

Sie können innerhalb einer Unterroutine eine weitere Unterroutine aufrufen. Unterroutinen dürfen nicht mehr als 16-fach verschachtelt werden.

Sie können mehr als eine RETURN-Anweisung in Ihre Unterroutine schreiben. Wenn Sie Bedingungen überprüfen, die den Programmlauf festlegen, werden Sie mehrere RETURN-Anweisungen in einer einzigen Unterroutine benötigen.

Anmerkung: Es wäre ratsam, bei einer GOSUB-Anweisung anstelle der Zeilennummern Sprungmarken-Namen anzugeben. Zeilennummern werden durch Verwendung der RENUM-Anweisung verändert. Deshalb müßten Sie nach Verwendung einer RENUM-Anweisung alle GOSUB <Zeilennummer>-Anweisungen dahingehend überprüfen, ob die angegebenen Zeilennummern sich noch auf die dazugehörigen Unterroutinen beziehen. Bei Verwendung von GOSUB <Sprungmarken-Name> werden alle Zeilenadressen von ST BASIC automatisch an die neuen Gegebenheiten angepaßt.

Beispiel:

```
Ok 10 GOSUB 100
Ok 20 REM RETURN-PUNKT DER UNTERROUTINE
Ok 30 PRINT A
Ok 40 END
Ok 100 REM ANFANG UNTERROUTINE
Ok 110 GOSUB BOO
Ok 120 A = 5*5
Ok 130 RETURN
Ok 140 BOO: PRINT "BOO!"
Ok 150 RETURN
Ok RUN
  15625
  BOO!
  25
Ok
```

Syntax: GOTO <Zeilennummer> oder GOTO <Sprungmarken-Name>

Effekt: Gibt die Programmkontrolle bedingungslos an eine angegebene Zeilennummer ab.

Erklärung:

Die GOTO-Anweisung übergibt die Programmkontrolle an eine angegebene Zeile und fährt dort mit der Programmausführung fort. Wenn Sie mit GOTO in eine nicht ausführbare Anweisung springen, beginnt die Programmausführung bei der nächsten ausführbaren Anweisung nach der angegebenen Zeile.

Anmerkung: Es wäre ratsam, bei einer GOTO-Anweisung anstelle der Zeilennummern Sprungmarken-Namen anzugeben. Zeilennummern werden durch Verwendung der RENUM-Anweisung verändert. Deshalb müßten Sie nach Verwendung einer RENUM-Anweisung alle GOTO <Zeilennummer>-Anweisungen dahingehend überprüfen, ob die angegebenen Zeilennummern sich noch auf die dazugehörigen Unterroutinen beziehen. Bei Verwendung von GOTO <Sprungmarken-Name> werden alle Zeilenadressen von ST BASIC automatisch an die neuen Gegebenheiten angepaßt.

Beispiel:

```
Ok 10 TOP: INPUT "BITTE NAMEN EINGEBEN"; NAME$
   .
   .
   .
Ok 100 INPUT "WOLLEN SIE DAS PROGRAMM BEENDEN";
     ANTWORT$
Ok 120 IF ANTWORT$ = "JA" THEN GOTO 200
Ok 130 GOTO TOP
Ok 200 END
```

Syntax: GOTOXY <Spaltenposition>,<Reihenposition>

Effekt: Setzt den Ausgabe-Cursor an den Schnittpunkt der angegebenen Reihe/Spalte.

Erklärung:

GOTOXY setzt den Ausgabe-Cursor an den Schnittpunkt der Reihe/Spalte, die durch die beiden Parameter vorgegeben wurde.

Beispiel:

```
Ok 10 GOTOXY 2,3
Ok 20 PRINT "SPALTE2, REIHE3"
```

Syntax: X = HEX$(numerischer Ausdruck)

Effekt: Gibt eine Zeichenkette aus, die dem Hexadezimalwert einer Zahl entspricht.

Erklärung:

Eine Hexadezimalzahl ist eine Basis 16 Integerzahl. Hexadezimalzahlen werden in den Zahlen 0 bis 9, gefolgt von den Zeichen A bis F dargestellt und repräsentieren die Werte 1 bis 15.

HEX$ stellt der ausgegebenen Hexadezimalzahl kein &H voran. Wollen Sie den Wert in einem Programm verwenden, müssen Sie ihm das Zeichen &H voranstellen, um kennzuzeichnen, daß es sich um einen hexadezimalen Wert handelt.

HEX$ rundet Realzahlen vor der Umwandlung in Hexadezimal in Integerzahlen auf oder ab.

Der normale gültige Bereich für Integerzahlen liegt zwischen -32768 und 65535.

Wenn Sie einen Adressen-Ausdruck einer Integer-Variablen zuordnen wollen, müssen Sie den Wert unter Verwendung von VAL der Variablen zuordnen, um einen Integer-Überlauffehler zu vermeiden (siehe auch nachstehendes Beispiel).

Beispiel:

```
Ok 10 A% = VAL("&H" + HEX$(FRE(0)))
Ok 20 PRINT A%
Ok RUN
-22536
Ok
```

Syntax: IF <logischer Ausdruck> THEN <Anweisung> <:Anweisung> [ELSE <Anweisung> <:Anweisung>]

Effekt: Stellt Bedingungen auf, die den Programmlauf festlegen.

Erklärung:

Die IF-Anweisung entspricht einem Ausdruck, der entweder wahr (nicht 0) oder falsch (0) ist. Ist der Ausdruck wahr, werden die Anweisungen hinter THEN ausgeführt. Ist er falsch, fährt das Programm bei der Anweisung hinter ELSE mit der Ausführung fort. Wurde kein ELSE angegeben, wird die Programmausführung in der nächsten ausführbaren Programmzeile wieder aufgenommen.

Sie können IF-Anweisungen innerhalb einer IF-Anweisung verwenden. Jede ELSE-Angabe bezieht sich auf das nächstliegende THEN. THEN- und ELSE-Bestimmungen haben nur in Verbindung mit einer IF- Anweisung Gültigkeit.

Sie können innerhalb der THEN- oder ELSE-Bestimmung einer IF- Anweisung eine FOR- oder WHILE-Schleife schreiben. Die FOR- oder WHILE-Anweisung muß sich vollständig innerhalb der THEN- oder ELSE-Bestimmung befinden: das dazugehörige NEXT muß sich in derselben Bestimmung wie die FOR-Anweisung, bzw. das entsprechende WEND in derselben Bestimmung wie die WHILE-Anweisung befinden (siehe auch nachstehendes Beispiel 1).

Wenn Sie eine IF-Anweisung innerhalb einer FOR- oder WHILE-Anweisung verwenden (wobei alle Segmente in derselben Anweisungszeile stehen müssen), schließt ein NEXT oder WEND auch die IF-Konstruktion (siehe auch nachstehendes Beispiel 2).

Beispiel 1:

```
Ok 5 A%=5
Ok 10 IF A%>3 THEN FOR K%=1 TO
    5:PRINT A%*K%: NEXT ELSE FOR
    K%=1 TO 5: PRINT A%/K%:NEXT
Ok RUN
  5
  10
  15
  20
  25
Ok
```

Beispiel 2:

```
Ok 10 FOR X = 1 TO 5:IF X<3 THEN PRINT X*X:NEXT:PRINT
"DONE"
Ok RUN
  1
  4
DONE
(Die NEXT-Bestimmung wird immer ausgeführt)
```

INP **X = INP (3)** **FUNKTION**

Syntax: X = INP (<Datenkanal-Nummer>)

Effekt: Gibt einen Byte-Wert von einem ausgewählten Datenkanal aus.

Erklärung:

Die Datenkanal-Nummer muß zwischen 0 und 65535 liegen. Die Funktion INP ist das Gegenstück zur OUT-Anweisung.

Um den Status des Datenkanals lesen zu können, wird ein negativer Datenkanal-Wert (z.B. INP(-3)) eingegeben. Eine Null zeigt an, daß kein Zeichen verfügbar ist; -1 signalisiert, daß ein Zeichen verfügbar ist.

Für den ATARI ST Computer gelten folgende Datankanal-Zuweisungen:

0 = PRINTER (Parallel-Port)
1 = AUX (RS-232)
2 = CONSOLE (Bildschirm)
3 = MIDI (Musical Instrument Digital Interface)
4 = KEYBOARD (Tastatur)

Beispiel:

```
Ok 200 Y = INP (3)
Ok 210 IF INP (3) > X THEN GOTO 200
```

INPUT **INPUT A$** **ANWEISUNG**
INPUT "NAME: ", A$
INPUT "NAME"; A$
INPUT X, Y, Z
INPUT "Hoehe, Breite, Alter", X, Y, Z

Syntax: INPUT [;] [<Prompt-String><; oder,>] <Variable> ,
<Variable<

Effekt: Ermöglicht eine Dateneingabe während des Programmlaufes und ordnet diese Daten den Programm-Variablen zu.

Erklärung:

Die INPUT-Anweisung bittet um eine Dateneingabe während der Programmausführung und erwartet Ihre Antwort. Nach erfolgter Eingabe muß die RETURN-Taste gedrückt werden, um die Eingabe an das Programm zu übermitteln.

Der Prompt-String ist eine String-Konstante und muß in Anführungszeichen gesetzt werden. Die Variablen können Zeichenketten oder Zahlen sein. Ihre Eingaben müssen in der passenden Variablenart erfolgen. Zeichenketten-Antworten werden nicht in Anführungszeichen gesetzt.

Wenn Sie einen Prompt-String verwenden, druckt die INPUT-Anweisung diese Zeichenkette als Anfrage auf den Bildschirm. Dabei wird der Prompt-String als Frage oder Aufforderung dargestellt, abhängig davon, ob Sie die Eingabe des Strings mit ei nem Komma oder einem Strichpunkt-Zeichen abgeschlossen haben.

Wird der Prompt-String mit einem Strichpunkt von den Variablen abgetrennt, fügt die INPUT-Anweisung am Ende des Prompt-Strings ein Fragezeichen, gefolgt von einer Leerstelle, an.

Trennen Sie den Prompt-String mit einem Komma von den Variablen ab, wird die Eingabe ohne Fragezeichen und ohne Leerstelle auf dem Bildschirm ausgegeben. Ihre Antwort wird in dieselbe Zeile eingegeben. Aus diesem Grund müssen Sie als letztes Zeichen in Ihrem Prompt-String eine Leerstelle eingeben, falls Sie einen Abstand zwischen der Anfrage und der Antwort auf dem Bildschirm wünschen.

Wenn Sie keinen Prompt-String bzw. einen Null-String schreiben, druckt INPUT ein Fragezeichen und eine Leerstelle auf den Bildschirm und wartet Ihre Antwort ab.

Die INPUT-Anweisung gibt eine Anfrage für jede Variable aus. Dabei entspricht jede Antwort einer INPUT-Variablen. Weicht die Anzahl der Variablen von der Anzahl der Antworten ab, tritt ein Fehler auf.

Sie müssen individuelle Antworten durch Kommata voneinander absetzen. Sie können auch innerhalb einer Antwort Kommata verwenden. Allerdings muß der Antwort-String dann in Anführungszeichen gesetzt werden.

Sie können als Antwort auf eine INPUT-Anfrage eine vollständige Zeile mit Zeichen eingeben. Eine Zeilenschaltung oder ein Zeilenumbruch schließt die Eingabezeile ab. Die maximale Zeilenlänge ist 255 Zeichen.

Beispiel:

```
Ok 10 INPUT "HEUTIGES DATUM EINGEBEN: ", X$
Ok 20 INPUT "KENNUMMER EINGEBEN: ",Z$
Ok 30 IF Z$ = "359152" THEN GOTO 100
Ok 40 PRINT "UNBEFUGTER DATENZUGRIFF NICHT ERLAUBT":
    END
Ok 100 PRINT "ZUGRIFF AUF DATEN GESTATTET!": END
Ok RUN
HEUTIGES DATUM EINGEBEN: 9 NOVEMBER 1985
KENNUMMER EINGEBEN: 359152
ZUGRIFF AUF DATEN GESTATTET!
Ok
```

Syntax: INPUT#<Dateinummer>, <Variable>, <Variable>

Effekt: Liest Daten aus einer sequentiellen Diskettendatei in Programm-Variablen ein.

Erklärung:

Die Dateinummer ist die Zahl, die Sie der Datei beim Öffnen zugewiesen hatten. Sie ordnen die Daten der Datei Variablen zu. Die Typen einer Variablen und der ihr zugeordneten Daten müssen übereinstimmen.

Die INPUT#-Anweisung arbeitet ähnlich wie die INPUT-Anweisung. Allerdings erscheint keine Meldung. Bevor Sie die eingegebenen Daten-Begriffe einer Variablen zuordnen, entfernt INPUT# alle vorangestellten Leerzeichen, Tabulatoren, Zeilenschaltungen und Zeilenvorschübe, die Sie zusammen mit den Daten eingegeben hatten. Das erste Zeichen nach den oben angeführten Sonderzeichen wird als Anfangspunkt der Daten angesehen. Ein Leerzeichen, eine Zeilenschaltung, ein Zeilenvorschub, ein Komma oder das Erreichen von 255 Zeichen signalisiert den Endpunkt der Daten.

Es gibt drei Arten von Daten für die INPUT#-Anweisung: Zahlen in allen numerischen Formaten, angeführte und nicht angeführte Strings.

Daten werden als Zahl angesehen, wenn die Variable, der sie zugeordnet werden, numerischen Charakter hat. Andernfalls werden sie als String behandelt. Zahlen werden durch Erreichen des Dateiendes bzw. nach 255 Zeichen, durch eine Zeilenschaltung, einen Zeilenvorschub, ein Komma oder ein nicht numerisches Zeichen beendet.

Strings werden als angeführt behandelt, wenn das erste Zeichen nach eventuellen Leerstellen ein Anführungszeichen ist. Alle zwischen zwei Anführungszeichen gesetzte Daten werden als Daten in angeführten Strings angesehen. Anführungszeichen dürfen innerhalb eines angeführten Strings nicht als reguläre Zeichen verwendet werden, da hierdurch fälschlicherweise das Ende des Strings markiert würde. Angeführte Strings werden ebenfalls durch Erreichen des Dateiendes bzw. nach 255 Zeichen beendet.

Nicht angeführte Strings können im Gegensatz zu angeführten Strings Anführungszeichen enthalten. Sie werden durch eine Zeilenschaltung, einen Zeilenvorschub, ein Komma oder durch Erreichen des Dateiendes bzw. nach 255 Zeichen beendet. Vorangestellte Leerzeichen in nicht angeführten Strings werden ignoriert.

Beispiel:

```
Ok 10 OPEN "I", #1, "RECHNUNG"
Ok 20 INPUT#1, KUNDE$, RECHNUNG%, DATUM$
```

INPUT$** **X$ = INPUT$ (6) **FUNKTION**
X$ = INPUT$ (6, #1)

Syntax: INPUT$(<Anzahl der Zeichen>[,[#]<Dateinummer<])

Effekt: Gibt die angegebene Zahl von Zeichen über Tastatur oder eine Daten-Datei aus.

Erklärung:

INPUT$ liest die angegebene Anzahl von Zeichen über Tastatur oder eine Datei aus und gibt einen String aus, der diese Zeichen enthält. Alle Zeichen werden ausnahmslos ohne Übersetzung und genau in der eingegebenen Form ausgegeben. So wird beispielsweise ein [CONTROL] [G] vom Terminal und ein [CONTROL] [Z] von der Daten-Datei an den String geleitet.

Wenn Sie den String aus einer Datei eingeben, müssen Sie eine geöffnete Dateinummer angeben. Versuchen Sie, nach dem Dateiende Daten auszulesen, erhalten Sie einen Fehler.

Lesen Sie hierzu auch unter EOF nach.

Beispiel:

```
Ok 20 X$ = INPUT$(6)
Ok 30 IF X$ = "GEORG" THEN 1000 ELSE PRINT "UNGUELTIG": END
Ok 1000 PRINT "GUELTIG"
Ok RUN
KENNWORT?
ARNOLD
UNGUELTIG
Ok
```

INSTR **X = INSTR (3,A$,"DO")** **FUNKTION**
X = INSTR (3,A$,B$)

Syntax: X = INSTR([<Anfangspunkt>,] <Zielstring-Ausdruck>, <Musterstring>)

Effekt: Sucht eine Zeichenkette innerhalb eines anderen Strings und gibt deren Position aus.

Erklärung:

INSTR sucht nach dem ersten Vorkommen eines Musterstrings innerhalb eines Zielstrings und gibt dessen Position aus.

Sie können einen Anfangspunkt für die Suche vorgeben. Der optionale Anfangspunkt ist eine Integerzahl zwischen 1 und 255.

Zielstring und Musterstring können String-Konstanten, Ausdrücke oder Variablen sein.

Ist der Musterstring länger als der Zielstring, oder ist der Zielstring ein Nullstring, oder kommt der Musterstring im Zielstring nicht vor, gibt INSTR 0 aus.

Ist der Musterstring 0, gibt INSTR die Anfangsposition Null aus.

Beispiel:

```
Ok 10 X$ = "WIE GEHT ES DIR?"
Ok 20 X = INSTR(3,X$,"GE")
Ok 30 PRINT X
Ok RUN
  5
Ok
```

INT **X=INT(Y)** **FUNKTION**

Syntax: X = INT(numerischer Ausdruck)

Effekt: Wandelt eine Zahl oder einen Ausdruck in eine Integerzahl um.

Erklärung:

INT entfernt Dezimalstellen.

Beispiel:

```
Ok 10 X = INT(2.999)
Ok 20 PRINT X
Ok RUN
  2
Ok
```

KILL **KILL DATEI.DAT** **ANWEISUNG**

Syntax: KILL<String-Ausdruck>

Effekt: Löscht eine Diskettendatei.

Erklärung:

Der String-Ausdruck entspricht einem Dateinamen. KILL löscht die Datei mit dem angegebenen Dateinamen. So löscht KILL A$ die Datei, die über A spezifiziert wurde. Sie können mit KILL jede Art von Diskettendatei löschen. Sie können eine Datei jedoch nicht mit KILL löschen, die derzeit geöffnet ist. In diesem Fall erhalten Sie eine Fehlermeldung.

Im nachstehenden Beispiel wird eine Datei mit dem Namen ATARI.BAS erstellt. Diese Datei wird dann über die KILL-Anweisung gelöscht.

Im Gegensatz zu ERA kann KILL auch innerhalb eines ST BASIC Programmes verwendet werden (z.B. Ok 10 KILL "DATEN.1").

Beispiel:

```
Ok NEW
Ok 10 A=45:B=56
Ok 20 PRINT A+B
Ok 30 END
Ok SAVE ATARI
Ok B $="ATARI.BAS"
Ok KILL B$
Ok
```

Syntax: X$ = LEFT$(<Zielstring><Anzahl der Zeichen>)

Effekt: Gibt eine Zeichenkette aus, die die ersten Zeichen eines Strings, gerechnet von links aus, enthält.

Erklärung:

LEFT$ beginnt beim ersten Zeichen von links und gibt die von Ihnen spezifizierte Anzahl von Zeichen, gezählt nach rechts, aus. Die Anzahl der Zeichen muß eine positive Zahl zwischen 1 und 255 sein. Real-Ausdrücke werden in Integerzahlen umgewandelt.

Der Zielstring kann eine String-Konstante, -Variable oder ein String-Ausdruck sein.

Ist die Anzahl der Zeichen größer als die Länge des Zielstrings, gibt LEFT$ den gesamten Zielstring aus. Ist die Anzahl der Zeichen 0, gibt LEFT$ einen Nullstring aus.

Beispiel:

```
Ok 10 INPUT "RADIUS";R
Ok 20 PRINT 3.1416*R^2
Ok 30 INPUT "NEUER BEREICH";C$
Ok 40 IF LEFT$(C$,1)="J"THEN 10
Ok 50 END
.
.
.
RADIUS ?3
28.2735
NEUER BEREICH ?J
RADIUS ?
```

Syntax: Z = LEN(<String-Ausdruck>)

Effekt: Gibt die Länge einer Zeichenkette aus.

Erklärung:

LEN gibt die Anzahl von Zeichen in einer Zeichenkette als Integerzahl aus. Ist der Ausdruck ein Nullstring, gibt LEN Null aus.

Beispiele:

```
Ok 10 ADDRESS$ = "2114 PARKER ST, BIRDLAND, NEW YORK"
Ok 20 FOR X = 1 TO LEN(ADDRESS$)
Ok 40 PRINT CHR$(42);
Ok 50 NEXT X
Ok RUN
**********************************
Ok

Ok 10 A$="DER STRING IST 30 ZEICHEN LANG"
Ok 20 PRINT A$
Ok 30 PRINT LEN(A$)
Ok RUN
DER STRING IST 30 ZEICHEN LANG
  30
```

Syntax: LET <Variable>=<Ausdruck>

Effekt: Ordnet einen Wert einer Variablen oder Array- Variablen zu.

Erklärung:

Die Verwendung von LET für die Zuordnung von Werten zu Variablen ist optional. So ist beispielsweise LET X = Y identisch zu X = Y. Sowohl Variable, als auch Ausdruck können Strings oder Zahlen sein. Bei numerischen Variablen und Ausdrücken wird die Art des Ausdruckes umgewandelt, um dem Variablentyp zu entsprechen.

Beispiel:

```
Ok 10 LET NAME$ = "BEYELSTEIN"
Ok 20 WOHNORT$ = "ELTVILLE, HESSEN"
Ok 30 LET REISEZIEL$ = HAWAII"
Ok 40 TAG.DER.ABREISE = 10.11.
Ok 50 TAG.DER.RUECKKEHR = 28.11.
Ok 60 DAUER.DER.REISE = TAG.DER.RUECKKEHR -
      TAG.DER.ABREISE
Ok 70 PRINT NAME$
Ok 80 PRINT WOHNORT$
Ok 90 PRINT "REISEZIEL: ": REISEZIEL$
Ok 100 PRINT "DAUER DER REISE: " DAUER.DER.REISE
Ok RUN
BEYELSTEIN
ELTVILLE, HESSEN
REISEZIEL: HAWAII
DAUER DER REISE: .18
Ok
```

Syntax: LINE INPUT[;] [<Prompt>[,oder ;]]<String-Variable>

Effekt: Erfordert eine Eingabe über Tastatur und ordnet diese Eingabe einer String-Variablen zu.

Erklärung:

LINE INPUT entspricht in etwa der INPUT-Anweisung, da hier eine Eingabe über Tastatur gefordert wird. LINE INPUT erlaubt jedoch die Eingabe einer vollständigen Zeile mit 255 Zeichen als Antwort. Ihre Antwort wird der String-Variablen zugeordnet. Eine Zeilenschaltung oder ein Zeilenvorschub schließt Ihre Eingabe ab und übermittelt sie dem Computer.

Die optionale Prompt-Angabe ist ein String, den Sie als Aufforderung für eine Eingabe schreiben können. LINE INPUT stellt diese Prompt-Angabe im Ausgabefenster dar und wartet auf Ihre Antwort. LINE INPUT ergänzt die Prompt-Angabe nicht automatisch mit einem Fragezeichen oder einer Leerstelle. Sie können jedoch selbst ein Fragezeichen oder eine Leerstelle innerhalb des Prompt-Strings eintragen. Das Einfügen einer Leerstelle ist ratsam, da ansonsten Ihre Eingabe direkt hinter die Prompt-Angabe gesetzt würde.

Beispiel:

```
Ok 10 LINE INPUT "GRUND FUER DIE RUECKSENDUNG";R$
   Ok 20 PRINT "DANKE! WIR BEARBEITEN IHRE RETOURE"
Ok RUN
GRUND FUER DIE RUECKSENDUNG?
FALSCHE GROESSE, FALSCHE FARBE, GEFAELLT NICHT.
DANKE! WIR BEARBEITEN IHRE RETOURE.
Ok
```

Syntax: LINE INPUT#<Dateinummer>, <String-Variable>

Effekt: Erfordert eine Eingabe über eine sequentielle Diskettendatei und ordnet diese Eingabe einer String-Variablen zu.

Erklärung:

Wie LINE INPUT ordnet auch LINE INPUT# eine Zeile mit maximal 254 Zeichen Länge einer String-Variablen als Eingabe zu. Allerdings kommt hier die Eingabe von einer sequentiellen Diskettendatei. Die Dateinummer ist die Zahl, die Sie der Datei beim Öffnen zugewiesen hatten.

LINE INPUT# liest alle Zeichen in einer sequentiellen Datei, bis es bei einer Zeilenschaltung angelangt. Dann werden diese Zeichen der String-Variablen zugeordnet. Die nächste LINE INPUT#- Anweisung beginnt am Endpunkt der ersten LINE INPUT#-Anweisung und ordnet die nachfolgende Zeile, wiederum bis zu einer Zeilenschaltung, der nächsten String-Variablen zu.

Folgt einem Zeilenvorschub direkt eine Zeilenschaltung, werden diese Zeichen als reguläre Zeichen behandelt und markieren kein Zeilenende.

Beispiel:

```
Ok 10 OPEN "O", #4, "PUNKTE"
Ok 20 LINE INPUT "ANGABE TEAMS, SIEGER UND PUNKTE.", S$
Ok 30 PRINT#4, S$
Ok 40 CLOSE #4
Ok 50 OPEN "I", #4, "PUNKTE"
Ok 60 LINE INPUT#4,S$
Ok 70 PRINT S$
Ok 80 CLOSE #4
Ok RUN
ANGABE TEAMS, SIEGER UND PUNKTE.
HSV & FCB: FCB. 3-0; HERTA BSC & FC KOELN: FC KOELN. 2-1
HSV & FCB: FCB. 3-0; BORUSSIA & FC KOELN: FC KOELN. 2-1
Ok
```

Syntax: LINEF [<Koordinatenpunkt, Koordinatenpunkt>]

Effekt: LINEF zeichnet eine Linie.

Erklärung:

LINEF zeichnet eine Linie zwischen den beiden angegebenen Koordinatenpunkten. Die Koordinatenpunkte sind Pixel-Positionen, die von der oberen linken Ecke (0,0) des Ausgabefensters aus gezählt werden. Die Anzahl verfügbarer Punkte in der Horizontalen und Vertikalen ist abhängig von der gewählten Auflösung.

Beispiel:

```
Ok 10 COLOR 1,0,1:CLEARW 2
Ok LINEF 50,50,80,80
Ok RUN
[Im Ausgabefenster erscheint eine Linie zwischen den beiden
Koordinatenpunkten]
  Ok
```

LIST **LIST** **BEFEHL**
LIST 10-50
LIST 10, 30, 50
LIST 10-30, 70-90
LIST - 30

Syntax: LIST [<Zeilenangabe, Liste>]

Effekt: Zeigt Programmzeilen im Auflistungsfenster an.

Erklärung:

LIST zeigt von Ihnen spezifizierte Zeilen des aktuellen Programmes im Auflistungsfenster an.

LIST listet das vollständige Programm auf.

LIST 10 zeigt lediglich Zeile 10 des Programmes an.

LIST 10-50 zeigt die Programmzeilen 10 bis 50 an.

LIST 10, 30, 50 zeigt die Zeilen 10, 30 und 50 des Programmes an.

LIST 10-30, 70-90 listet zwei Gruppen von Zeilen auf, einmal Zeile 10 bis 30, und einmal Zeile 70 bis 90.

LIST - 30 listet alle Zeilen vom Anfang des Programmes bis zur Zeile 30 auf.

Durch Betätigen von [CONTROL] [G] beenden Sie das Auflisten und kehren zurück ins Befehlsfenster.

LLIST **LLIST** **BEFEHL**
LLIST 10-50
LLIST 10, 30, 50
LLIST 10-30, 70-90
LLIST - 30

Syntax: LLIST [<Zeilenangabe, Liste>]

Effekt: LLIST listet das Programm auf dem Drucker auf.

Erklärung:

LLIST arbeitet genauso wie LIST; allerdings werden die angegebenen Zeilen über den Drucker ausgegeben.
Der Befehl WIDTH LPRINT stellt die Zeilenbreite für den Drucker ein. ST BASIC setzt die Zeilenbreite automatisch auf 72 Zeichen fest. WIDTH LPRINT 40 würde diese Voreinstellung auf 40 Zeichen pro Zeile verändern.

LOAD **LOAD DATEI** **BEFEHL**

Syntax: LOAD <Dateiname>

Effekt: LOAD lädt Programmdateien in den Arbeitsspeicher.

Erklärung:

LOAD lädt ST BASIC Programmdateien in den Arbeitsspeicher des Computers. LOAD nimmt einen .BAS Extender an, sofern Sie keine anderweitigen Vorgaben festlegen. Wenn Sie ein Programm mit LOAD in den Arbeitsspeicher laden, wird ein noch im Speicher befindliches Programm mit sämtlichen Variablen dadurch gelöscht.

LOAD entspricht dem Befehl OLD.

LOC **X = LOC(1)** **FUNKTION**

Syntax: X = LOC(<Dateinummer>)

Effekt: Gibt entweder eine Datensatz-Nummer, oder die Anzahl der Bytes, die von einer Datei gelesen, bzw. in eine Datei geschrieben wurden, aus.

Erklärung:

Bei Verwendung nach GET oder PUT für eine Random-Diskettendatei gibt LOC die Nummer des Datensatzes aus, die zuletzt mit GET oder PUT gelesen oder geschrieben wurde.

```
GET #1
PUT #1,LOC(1)
```

ersetzt den Datensatz #1 in dem Slot, aus dem er gelesen wurde.

Bei Verwendung mit sequentiellen Dateien gibt LOC die Anzahl von Bytes aus, die seit dem Öffnen der Datei gelesen oder geschrieben wurden.

Beispiel:

```
Ok 10 OPEN "R", #8, "DATEI"
Ok 20 FIELD #8, 20 AS Z$, 3 AS V$
Ok 30 GET #8, C%
Ok 40 IF LOC(8) > 25 THEN GOTO 90
```

LOF **X=LOF(I)** **FUNKTION**

Syntax: X = LOF(<Dateinummer>)

Effekt: Gibt die Anzahl der Bytes in der Datei aus.

Erklärung:

Bei einer Datei, die gerade für eine Ausgabe geöffnet wurde, entspricht die Anzahl der Bytes 0.

Beispiel:

```
Ok 100 X = LOF(#5)
Ok 110 IF X > 100 THEN PRINT "OEFFNE NEUE DATEI": GOTO 200
```

LOG **X=LOG(N)** **FUNKTION**

Syntax: X = LOG(<numerischer Ausdruck>)

Effekt: Gibt den natürlichen Logarithmus einer Zahl aus.

Erklärung:

Der numerische Ausdruck muß größer als Null sein.

Beispiel:

```
Ok 10 PRINT LOG(23)/LOG(2)
Ok RUN
  4.52356
Ok
```

LOG10 **X = LOG10(Y)** **FUNKTION**

Syntax: X = LOG10(<numerischer Ausdruck>)

Effekt: Gibt den Basis 10 - Logarithmus einer Zahl aus.

Erklärung:

Der numerische Ausdruck muß größer als Null sein.

Beispiel:

```
Ok 10 X = LOG10(1000)
Ok 20 PRINT X
Ok RUN
  3
```

LPOS **LPOS(X)** **FUNKTION**

Syntax: LPOS(X)

Effekt: Gibt die Position des Druckkopfes des Zeilendruckers innerhalb des Zeilendrucker-Speichers aus.

Erklärung:

Die ausgegebene Position ist Anzahl der Zeichen, die seit der letzten Zeilenschaltung gedruckt wurden. Die Rücktaste wird mit -1 gezählt. Falls Sie Drucker-Kontrollzeichen verwenden, durch die die Position des Druckkopfes verändert wird, kann LPOS die genaue Position des Druckkopfes nicht reflektieren.

Beispiel:

```
Ok 10 X = 90
Ok 20 IF LPOS(X) > 45 THEN GOTO 100
```

Syntax: LPRINT [<Liste mit Ausdrücken>]
LPRINT USING <Formatstring-Ausdruck>;<Liste mit Ausdrücken>

Effekt: Leitet die Ausgabe an den Drucker.

Erklärung:

Die LPRINT-Anweisung arbeitet wie die PRINT- und PRINT USING- Anweisungen. Allerdings geht hier die Ausgabe an einen Zeilendrucker. Sie können die Zeilenbreite für den Drucker über die WIDTH LPRINT-Anweisung einstellen. Die Voreinstellung liegt bei 72 Zeichen pro Zeile. Der Formatstring-Ausdruck muß durch einen Strichpunkt von der Variablen-Liste abgetrennt werden. Die aufgelisteten Ausdrücke müssen durch Kommata voneinander abgesetzt sein.

Lesen Sie hierzu auch unter WIDTH LPRINT nach.

Beispiel:

```
Ok 10 LPRINT "DIESE AUSGABE ERFOLGT AN DEN DRUCKER"
```

Syntax: LSET<String-Variable>=<String-Ausdruck>

Effekt: Verschiebt einen String in eine spezifizierte String-Variable, ohne die String-Variable neu zuzuordnen.

Erklärung:

LSET wird normalerweise dazu verwendet, um Daten in Datei- Speicher zu übertragen. Dazu werden die Daten in Variable übertragen, die über eine vorhergegangene FIELD-Anweisung in Datei-Speichern aufgezeichnet wurden. LSET ist in seinen Verwendungsmöglichkeiten allerdings nicht allein darauf beschränkt.

Belegt der String-Ausdruck eine geringere Anzahl an Bytes, als Sie der String-Variablen in einer FIELD-Anweisung zugeteilt hatten, justiert LSET den linken Rand und verschiebt den String durch Einsetzen von Leerstellen weiter nach rechts.

Ist der String länger als die Ziel-Stringvariable, werden die zusätzlichen Zeichen von LSET ignoriert.

Belegt ein String eine größere Anzahl an Bytes, als Sie ihm in der FIELD-Anweisung zugeteilt haben, werden die rechts liegenden Zeichen entfernt.

Sie müssen Zahlen und numerische Variable mit MKD$, MKS$ oder MKI$ in Strings umwandeln, bevor Sie sie mit LSET verschieben.

Das Gegenstück zu LSET ist RSET.

Beispiel:

```
Ok 10 OPEN "I", #2, "TEST", 5
Ok 20 FIELD #2, 5 AS S$
Ok 30 LSET N$ = NN$
```

Syntax: MERGE <Dateiname>

Effekt: Fügt eine ST BASIC Diskettendatei in ein Programm im Arbeitsspeicher ein.

Erklärung:

Der MERGE-Befehl fügt eine Datei von Diskette in die bereits im Arbeitsspeicher befindliche Datei ein. Solange die Zeilennummern der beiden Dateien unterschiedlich sind, löscht MERGE die Originaldatei nicht. Stimmen dagegen Zeilennummern der Diskettendatei mit Zeilennummern der Datei im Speicher überein, werden die Programmzeilen im Arbeitsspeicher gegen die gleichlautenden Zeilennummern der Diskettendatei ersetzt.

Lesen Sie hierzu auch unter CHAIN nach.

Beispiel:

```
Ok 10 PRINT "DAS IST DAS ORIGINALPROGRAMM"
Ok 20 PRINT "DIESE ZEILE WIRD DURCH MERGE GELOESCHT"
Ok 30 PRINT "DIESE ZEILE BLEIBT WEGEN IHRER
UNTERSCHIEDLICHEN ZEILENNUMMER ERHALTEN"
Ok SAVE ORIGINAL
Ok NEW
Ok 15 PRINT "DAS IST DAS OVERLAY"
Ok 20 PRINT "DIESE ZEILE ERSETZT ZEILE 20 IM ORIGINAL"
Ok SAVE OVERLAY
Ok LOAD ORIGINAL
Ok MERGE OVERLAY
Ok RUN
DAS IST DAS ORIGINALPROGRAMM
DAS IST DAS OVERLAY
DIESE ZEILE ERSETZT ZEILE 20 IM ORIGINAL
DIESE ZEILE BLEIBT WEGEN IHRER UNTERSCHIEDLICHEN
ZEILENNUMMER ERHALTEN
Ok
```

Syntax: MID$(<String-Ausdruck>,<Anfangspunkt>[,Länge]) = (<String-Ausdruck>)

Effekt: Funktion: Gibt ein Segment einer Zeichenkette aus.
Anweisung: Ordnet einem String-Segment einen Wert zu.

Erklärung:

MID$ gibt ein Segment eines Strings aus. Der Anfangspunkt ist ein numerischer Ausdruck, der auf den Anfang des Segmentes zeigt. Die Länge ist ein numerischer Ausdruck, der die Länge des Segmentes rechts neben dem Anfangspunkt spezifiziert. Wenn Sie die Längenangabe entfallen lassen, gibt MID alle Zeichen hinter dem Anfangspunkt aus.

Ist die Angabe für den Anfangspunkt höher als die Stringlänge, gibt MID$ einen Nullstring aus.

Falls die Länge des Segmentes größer ist als die Anzahl von Zeichen rechts neben dem Anfangspunkt, werden alle Zeichen hinter dem Anfangspunkt ausgegeben.

MID$ kann auch dazu verwendet werden, um ein String-Segment zu definieren.

Lesen Sie hierzu auch unter RIGHT$ und LEFT$ nach.

Beispiel:

```
Ok 10 X$ = "MR. JAMES GRAHAM SCOTT"
Ok 20 Y$ = MID$(X$,18,5)
Ok 30 PRINT Y$
Ok RUN
SCOTT
Ok
```

Syntax: X$ = MKD$(<numerischer Ausdruck>)
X$ = MKI$(<Integerwert>)
X$ = MKS$(<numerischer Ausdruck>)

Effekt: Die Funktionen MKD$, MKI$ und MKS$ wandeln ASCII-Strings, die Zahlen repräsentieren, in Byte-Strings für die Verwendung in Random-Dateispeichern um.

Erklärung:

MKI$ gibt einen 2-Byte String aus.
MKS$ gibt einen 4-Byte String aus.
MKD$ gibt einen 8-Byte String aus.

Sie müssen ASCII-Werte mit diesen Funktionen in Zeichenketten umwandeln, bevor Sie sie mit RSET oder LSET in einen Random-Dateispeicher übertragen können. Die Funktionen CVD, CVI und CVS sind die Gegenstücke zu MKD$, MKI$ und MKS$.

Beispiel:

```
Ok 100 FINAL = (100/X) * (100 - Y)
Ok 110 FIELD #2, 5 AS Z$, 5 AS B$
Ok 120 LSET Z$ = MKI$(FINAL)
Ok 130 LSET B$ = T$
Ok 140 PUT # 2
  .
  .
  .
```

NAME **NAME AUG.DAT AS SEPT.DAT** **ANWEISUNG**

Syntax: NAME <alter String-Ausdruck> AS <neuer String- Ausdruck>

Effekt: Benennt eine Datei neu.

Erklärung:

Die NAME-Anweisung gibt einer bereits bestehenden Datei lediglich einen neuen Namen. NAME verändert weder die Datei, noch den Disketten-Inhalt. Vergewissern Sie sich, daß die alte Datei wirklich auf der Diskette vorhanden ist und der neue Name nicht bereits für eine andere Datei verwendet wurde. Ansonsten würde ein Fehler auftreten.

Beispiel:

```
Ok NAME "VERSION2.BAS" AS "VERSION3.BAS"
```

NEW **NEW NEUPRG.BAS** **BEFEHL**

Syntax: NEW [NAME]

Effekt: Löscht eine Datei im Arbeitsspeicher und benennt optional das neue Programm.

Erklärung:

Verwenden Sie NEW, wenn Sie beabsichtigen, ein neues Programm zu schreiben. Falls Sie das derzeit im Arbeitsspeicher befindliche Programm nicht gespeichert haben, wird dieses durch NEW gelöscht. Wenn Sie die Option NAME verwenden, können Sie den SAVE-Befehl später ohne Namensangabe verwenden.

Beispiel:

```
Ok 10 X = SQR(25)
Ok 20 PRINT X
Ok NEW
Ok LIST
Ok
```

Syntax: NEXT [<Zähler>] ,Zähler

Effekt: Markiert das Ende einer FOR/NEXT-Schleife.

Erklärung:

Die NEXT-Anweisung in einer FOR/NEXT-Schleife übergibt die Programmkontrolle an den Schleifenanfang. Die Schleife wird erneut durchlaufen, wenn die Zähl-Variable noch nicht größer ist als die in der FOR-Anweisung vorgegebene Obergrenze.

Die Angabe des Namens für die Zähl-Variable ist optional. Die NEXT-Anweisung nimmt die nächstliegende Zähl-Variable an.

Haben Sie Schleifen verschachtelt, müssen Sie angeben, zu welcher Zähl-Variablen Sie am Ende der Schleifenausführung zurückkehren wollen. Verwenden Sie NEXT, um die Programmausführung zuerst an die verschachtelte Schleife, und danach an die äußere Schleife zu übergeben. Hierzu wird als Erstes die verschachtelte Zähl-Variable, und danach die äußere Zähl-Variable angegeben.

Lesen Sie hierzu auch unter FOR nach.

Beispiel:

```
Ok 10 FOR Z = 1 TO 3
Ok 20 PRINT "Y"
Ok 30 FOR Q = 1 TO 2
Ok 40 PRINT "X"
Ok 50 NEXT Q,Z
Ok RUN
Y
X
X
Y
X
X
Y
X
X
Ok
```

OCT$** **X$ = OCT$ (Y) **FUNKTION**

Syntax: X$ = OCT$ (<numerischer Ausdruck>)

Effekt: Gibt den String-Ausdruck einer Basis 8-Zahl aus.

Erklärung:

OCT$ gibt eine Zeichenkette aus, die dem Basis 8 – Wert eines Hexadezimal- oder Dezimalwertes entspricht. Der Wert des Dezimal- oder Hexadezimalausdruckes wird auf eine Integerzahl gerundet, bevor er umgewandelt wird. Er muß zwischen −32768 und 32767 liegen.

Lesen Sie hierzu auch unter HEX$ und STR$ nach.

Beispiel:

```
Ok 10 X$ = OCT$(3.4)
Ok 20 PRINT X$
Ok RUN
3
```

OLD **OLD TEST** **BEFEHL**

Syntax: OLD <Dateiname>

Effekt: Lädt eine bestehende Programm-Datei in den Arbeitsspeicher. OLD ist identisch mit LOAD.

Erklärung:

OLD schließt alle geöffneten Dateien und löscht alle Variablen oder Daten im Arbeitsspeicher, bevor die angegebene Datei von Diskette in den Speicher geladen wird. OLD löscht alle ST BASIC Programme im Arbeitsspeicher.

Der Dateiname ist der Name, den Sie der Datei beim Speichern zugewiesen hatten. Sie müssen dabei den Dateityp .BAS nicht angeben.

Beispiel:

```
Ok OLD TEST
Ok
Das Programm TEST.BAS befindet sich nun im Arbeitsspeicher
```

Syntax: ON <numerischer Ausdruck> GOTO <Zeilenangabe> <Zeilenangabe>
ON <numerischer Ausdruck> GOSUB <Sprungmarke> , <Sprungmarke>

Effekt: Übergibt die Programmkontrolle an eine Programmzeile in einer Auflistung, abhängig vom errechneten Ergebnis des numerischen Ausdruckes. Die ON-Anweisung hat zwei Formen.

Erklärung:

Der Wert des numerischen Ausdruckes legt fest, wohin die Programmausführung übergeben wird. Entspricht der Wert des Ausdruckes 1, übergibt ON die Kontrolle an die erste Sprungmarke. Entspricht er 2, wird die Kontrolle entsprechend an die zweite Sprungmarke übergeben, usw.

Überprüfen Sie den Wert des Ausdruckes, bevor Sie eine ON- Anweisung schreiben.

Nicht-Integerzahlen werden auf die nächste ganze Zahl auf- oder abgerundet.

In einer ON GOSUB-Anweisung muß jeder numerische Ausdruck die Zahl der ersten Zeile einer Unterroutine sein. Die RETURN- Anweisung in der Unterroutine übergibt die Programmkontrolle an die erste ausführbare Anweisung, die der ON-Anweisung folgt.

Sie können in einer ON-Anweisung jede gültige Zeilenangabe verwenden. Eine ON-Anweisung kann an jeder beliebigen Stelle im Programm geschrieben werden.

```
10 ON X GOTO 200, PAINT, 400
```

Falls der Wert von X 1 ist, springt das Programm in Zeile 200; ist der Wert 2, springt das Programm in die Anweisung mit der Sprungmarke PAINT, usw.

Beispiel:

```
Ok 10 X = 1
Ok 20 ON X GOTO 70,80,90,990
Ok 70 PRINT "MONAT DES JAHRES:"X + 1
Ok 80 PRINT "MONAT DES JAHRES:"X + 2
Ok 90 PRINT "MONAT DES JAHRES:"X + 3
Ok 120 X=X+1: GOTO 20
Ok 990 END
Ok RUN
MONAT DES JAHRES: 2
MONAT DES JAHRES: 3
MONAT DES JAHRES: 4
MONAT DES JAHRES: 4
MONAT DES JAHRES: 5
MONAT DES JAHRES: 6
Ok
```

Syntax: ON ERROR GOTO <Zeilenangabe>

Effekt: Ermöglicht die Aufdeckung eines Laufzeit-Fehlers und übergibt die Kontrolle an eine Zeilennummer, sobald ein Fehler auftritt.

Erklärung:

ON ERROR GOTO springt in eine angegebene Programmzeile, sobald ST BASIC einen Fehler entdeckt, und ermöglicht dadurch die Handhabung von Laufzeitfehlern. Als Parameter muß eine Zeilennummer verwendet werden. Die Angabe einer Sprungmarke ist nicht möglich.

Sie können diese Fehlerbehandlung deaktivieren oder die ursprüngliche Fehlerbehandlung von ST BASIC wieder einrichten, wenn Sie ON ERROR GOTO 0 verwenden.

Wenn Sie ON ERROR GOTO 0 in einer Fehleraufdeckungsroutine verwenden, druckt ST BASIC seine Original-Fehlermeldung aus und hält das Programm an. Sie sollten in einer Fehleraufdeckungsroutine immer ON ERROR GOTO 0 verwenden, um unerwartete Fehler feststellen zu können.

Lesen Sie hierzu auch unter RESUME, sowie in Anhang D, Fehlermeldungen, nach.

Beispiel:

```
Ok 80 ON ERROR GOTO 100
```

Syntax: OPEN <Modus>,[#]<Dateinummer>,<Dateiname> [,<Datensatzlänge>]

Effekt: Ermöglicht die Ein- oder Ausgabe an eine Datei oder einen Datenkanal.

Erklärung:

Sie müssen eine Diskettendatei mit OPEN öffnen, bevor Sie Daten daraus entnehmen, bzw. in sie einlesen können. Die OPEN-Anweisung ordnet der Datei einen I/O-Speicher zu und legt den Modus fest, unter dem für eine Ein- und Ausgabe Zugriff auf die Datei genommen werden kann.

Die Dateinummer ist ein Integer-Ausdruck mit einem Wert zwischen 1 und 15. Eine Dateinummer ist einer Datei zugeordnet, solange diese geöffnet ist. Das Schließen einer Datei löscht die zugeordnete Dateinummer. Damit kann diese Nummer neu verwendet werden. Die Datensatzlänge ist ein Integer-Ausdruck, über den die Datensatzlänge für Random-Dateien festgelegt wird. Diese Angabe ist optional. Die vorgegebene Länge ist 128 Bytes. Die Angabe einer Datensatzlänge für sequentielle Dateien wird nicht beachtet.

Der Datei-Modus ist entweder eine sequentielle Ausgabe/Eingabe oder eine Random-Eingabe/Ausgabe. Der Modus wird durch Eingabe einer der nachfolgenden Kennbuchstaben festgelegt:

O Ausgabe für sequentielle Dateien
I Eingabe für sequentielle Dateien
R Ein- und Ausgabe für Random-Dateien

Die Eingabe dieser Kennbuchstaben muß in Großschreibung erfolgen.

Wenn Sie Random Access-Datensätze eingeben, muß die erste Datensatz-Nummer mit "1" eingegeben werden. Alle nachfolgenden Datensatz-Nummern müssen sequentiell sein. D.h., die erste Datensatz-Nummer ist "1", die Nummer für den zweiten Datensatz ist "2", für den dritten "3" usw. Diese Eingabe kann über eine FOR/NEXT-Schleife erfolgen. Datensätze, die in einer falschen Reihenfolge eingegeben werden, verursachen einen Fehler. Sobald die Datei etabliert ist, können die Datensätze (mit GET #1,VAR) in jeder beliebigen Reihenfolge aufgerufen werden.

Beispiel:

```
Ok 10 OPEN "R", #1, "GUTHABEN"
Ok 20 FIELD #1,10 AS V$, 10 AS X$,30 AS N$
Ok 30 INPUT "4-STELLIGEN CODE EINGEBEN", CODE!
Ok 40 GET #1, CODE!
```

Syntax: OPENW <Fenster-Nummer>

Effekt: Öffnet ein ST BASIC-Fenster.

Erklärung:

OPENW wird verwendet, um ein ST BASIC-Fenster zu öffnen, das zuvor über den Befehl CLOSEW geschlossen worden war. Das geöffnete Fenster wird im Vordergrund des Bildschirmes dargestellt. Wurde das Fenster bereits geöffnet, verbleibt es als oberstes Fenster auf dem Bildschirm. Die <Fenster-Nummer> spezifiziert die ST BASIC-Fenster wie folgt:

0 = Bearbeitungsfenster
1 = Auflistungsfenster
2 = Ausgabefenster
3 = Befehlsfenster

Anmerkung: OPENW gibt eine interne Meldung an den BASIC-Interpreter, durch die das System den Status der Fenster nachvollziehen kann. Aus diesem Grund sollten Sie ST BASIC-Fenster (die über CLOSEW geschlossen wurden) nicht über Direktaufrufe von AES öffnen.

OPTION BASE **OPTION BASE 0**
OPTION BASE 1 **ANWEISUNG**

Syntax: OPTION BASE <1 oder 0>

Effekt: Setzt die Basis für Array-Dimensionierungen.

Erklärung:

OPTION BASE wird für die Festsetzung des Mindestwertes für Array-Unterbereiche innerhalb einer Dimensionierung verwendet. Die vorgegebene Basis ist Null. Aus diesem
Grund hat das erste Element in einem Array einen Unterbereich Null. Sie können die Array-Dimensionierungen so setzen, daß sie bei 1 beginnen, oder sie auf Null belassen.

Sie können OPTION BASE beliebig oft verwenden.

Lesen Sie hierzu auch unter DIM nach.

Beispiel:

```
Ok 10 OPTION BASE 1
Ok 20 DIM A%(10)
Ok 30 OPTION BASE 0
Ok 40 DIM B%(10)
```

A% hat nun 10 Elemente (1-10) und B% 11 Elemente (0-10).

OUT **OUT 2, X** **ANWEISUNG**

Syntax: OUT <Integer-Ausdruck>,<Integer-Ausdruck>

Effekt: Übermittelt ein Byte an einen Datenausgabekanal.

Erklärung:

Der erste Integer-Ausdruck ist die Datenkanal-Nummer. Der zweite Ausdruck ist das Byte, das Sie an den Ausgabekanal leiten wollen. Der Wert des Bytes muß zwischen 0 und 65535 liegen.

Die Datenkanäle des ATARI ST Computers lauten wie folgt:

0 = PRINTER (Parallel-Port)
1 = AUX (RS-232)
2 = CONSOLE (Bildschirm)
3 = MIDI (Musical Instrument Digital Interface) 4 = KEYBOARD (Tastatur)

Beispiel:

```
Ok 100 If X% >5 THEN OUT 3,(X-2)
```

Syntax: PCIRCLE <horizont. Mittelpunkt, vertik. Mittelpunkt, Radius> [<,Anfangswinkel, Endwinkel>]

Effekt: PCIRCLE zeichnet ausgefüllte Kreise und Kreisausschnitt-Formen.

Erklärung:

PCIRCLE zeichnet vollständig in einer Farbe und mit einem Muster ausgefüllte Kreise. Der Kreismittelpunkt liegt am Schnittpunkt der beiden ersten Parameter (horizontaler und vertikaler Mittelpunkt). Die Positionen werden in Pixel angegeben, gerechnet von der oberen linken Ecke des Ausgabefensters.

Der dritte Parameter, Radius, wird ebenfalls in Pixel angegeben. Die horizontale und vertikale Pixelanzahl ist abhängig von der gewählten Auflösung. Der Kreis wird in der angegebenen FILL-Farbe (Parameter 2 der COLOR-Anweisung) gezeichnet.

Die letzten beiden Parameter, Anfangs- und Endwinkel, sind optional verwendbar. Werden sie nicht angegeben, zeichnet PCIRCLE einen vollständigen Kreis. Bei Angabe eines Anfangs- und Endwinkels wird ein Kreisausschnitt gezeichnet, der zwischen den beiden Punkten liegt. PCIRCLE zeichnet ein ausgefülltes Kreissegment und keinen Kreisbogen. Die Winkel werden in Grad mal 10 angegeben. So werden 45 Grad als 450, 180 Grad als 1800, usw. angegeben. 0 Grad zeigt im Ausgabefenster nach rechts, 90 Grad nach oben, 180 Grad nach links und 270 Grad nach unten. COLOR 1,3,1:PCIRCLE 100,30,30,0,3600 zeichnet einen vollständigen, grün ausgefüllten Kreis.

Lesen Sie hierzu auch unter CIRCLE, ELLIPSE und PELLIPSE nach.

Beispiel:

```
Ok 10 COLOR 1,0,1:CLEARW 2
Ok 20 CIRCLE 100,50,40
Ok 30 COLOR 1,2,1
Ok 40 PCIRCLE 100,50,40,300,900
Ok RUN
[Im Ausgabefenster erscheint ein schwarzer Kreis mit einem rot ausgefüllten
Kreissegment, beginnend bei 30 Grad, über eine Länge von 60 Grad]
Ok
```

Syntax: X = PEEK (<Speicheradresse>)

Effekt: Gibt den Inhalt einer Speicheradresse aus.

Erklärung:

PEEK gibt das an der angegebenen Speicheradresse befindliche Byte aus. Der Typ des ausgegebenen Wertes ist wie folgt von der letzten DEF SEG-Anweisung abhängig:

> Ist DEF SEG > 0, gibt PEEK ein Byte aus, unabhängig davon, wie die PEEK-Adresse spezifiziert wurde. Die in PEEK angegebene Adresse wird durch den Wert, der in der letzten DEF SEG-Anweisung angegeben wurde, eingerichtet.
>
> Wenn DEF SEG gleich Null ist, gibt PEEK ein 2-Byte Wort aus, sofern die PEEK-Adresse als FLOAT-Ausdruck angegeben wurde.
>
> Falls DEF SEG gleich Null ist und gleichzeitig die Adresse durch DEFDBL spezifiziert wurde, gibt PEEK einen 4-Byte Long-Integerwert aus.

Sie müssen die Speicheradresse über eine Variable, und nicht über eine Konstante spezifizieren (siehe nachstehendes Beispiel).

Lesen Sie hierzu auch unter POKE und SEG nach.

Anmerkung: Beim PEEKen wird der ST Computer in den Supervisor-Modus umgeschaltet. D.h., Sie können auf jede Speicheradresse, also auch auf geschützte Speicherplätze, Zugriff nehmen.

Beispiel:

```
Ok 100 BYTE% = PEEK(234)
```

Syntax: PELLIPSE <horizont. Mittelpunkt,vertik. Mittelpunkt, horizont.Radius, vertik. Radius>[<,Anfangswinkel,Endwinkel>]

Effekt: PELLIPSE zeichnet ausgefüllte Ellipsen und elliptische Kreisformen.

Erklärung:

PELLIPSE zeichnet eine Ellipse, deren Mittelpunkt durch die beiden ersten Parameter (horizontaler und vertikaler Mittelpunkt) festgelegt wird. Die Positionen werden in Pixel ausgedrückt, gerechnet von der oberen linken Ecke des Ausgabefensters. Parameter 3 und 4, horizontaler und vertikaler Radius, werden ebenfalls in Pixel angegeben. Die horizontale und vertikale Pixelanzahl ist abhängig von der gewählten Auflösung. Die Ellipse wird in der vorgegebenen Zeichenfarbe (Parameter 3 der COLOR-Anweisung) gezeichnet.

Die letzten beiden Parameter, Anfangs- und Endwinkel, sind optional verwendbar. Werden sie nicht angegeben, zeichnet PELLIPSE eine vollständige Ellipse. Bei Angabe eines Anfangs- und Endwinkels wird ein Ellipsenausschnitt gezeichnet, der zwischen den beiden Punkten liegt. PELLIPSE zeichnet ein ausgefülltes Kreissegment und keinen Kreisbogen.

Die Winkel werden in Grad mal 10 angegeben. So werden 45 Grad als 450, 180 Grad als 1800, usw. angegeben. 0 Grad zeigt im Ausgabefenster nach rechts, 90 Grad nach oben, 180 Grad nach links und 270 Grad nach unten. COLOR 1,3,1:PELLIPSE 100,50,50,50,0,3600 zeichnet eine vollständige, grün ausgefüllte Ellipse.

Lesen Sie hierzu auch unter ELLIPSE, CIRCLE und PCIRCLE nach.

Beispiel:

```
Ok 10 COLOR 1,0,1:CLEARW 2
Ok 20 PELLIPSE 100,80,40,80
Ok 30 COLOR 1,2,1
Ok 40 PELLIPSE 100,80,40,80,300,900
Ok RUN
[Auf dem Bildschirm wird eine schwarze Ellipse mit einem roten Kreisbogen
über 60 Grad, beginnend bei 30 Grad, gezeichnet]
Ok
```

Syntax: POKE<Adresse zum POKEn>,<Daten zum POKEn>

Effekt: Schreibt POKE-Daten in den Speicher.

Erklärung:

POKE speichert einen Wert der Daten, die mit POKE in eine Speicher-Adresse gebracht werden sollen. Die POKE-Adresse ist eine absolute Adresse, die als numerischer Ausdruck angegeben wird. Der Datentyp wird durch die letzte DEF SEG-Anweisung, sowie durch die Art der Spezifikation der POKE-Adresse definiert.

Die in POKE angegebene Adresse wird durch den Wert, der in der letzten DEF SEG-Anweisung angegeben wurde, eingerichtet.

Wenn DEF SEG gleich Null ist, entsprechen die Daten einem 2-Byte-Wort, sofern die POKE-Adresse als FLOAT-Ausdruck angegeben wurde.

Falls DEF SEG gleich Null ist und gleichzeitig die Adresse durch DEFDBL spezifiziert wurde, sind die Daten ein 4- Byte Long-Integerwert.

Liegt der Daten-Ausdruck außerhalb des gültigen Bereiches von 0 bis 255, speichert POKE das niederwertige Byte des Ergebnisses. So hat beispielsweise

```
Ok 5 DEF SEG=300000
Ok 10 POKE X%,257
```

den gleichen Effekt wie

```
Ok 5 DEF SEG=300000
Ok 10 POKE X%,1
```

Das Gegenstück zu POKE ist PEEK. sie können PEEK und POKE für die Übermittlung von Argumenten und Daten in Maschinensprache-Unterroutinen verwenden.

Lesen Sie hierzu auch unter PEEK und DEF SEG nach.

Beispiel:

```
Ok 100 FOR LOC%=1 TO LEN(OUT,MSG$)
Ok 120 POKE MSG.LOC%+LOC%,ASC(MID$(OUT,MSG$,LOC$,))
Ok 130 NEXT LOC%
```

Anmerkung: Beim PEEKen und POKEn wird der 520ST Computer in den Supervisor-Modus umgeschaltet. D.h., Sie können auf jede Speicheradresse, also auch auf geschützte Speicherplätze, Zugriff nehmen.

Das System stürzt ab, wenn Sie in Speicherplätze POKEn, die vom TOS Betriebssystem belegt sind. Bei einem Systemabsturz müssen Sie neu booten.

Syntax: X = POS(<Test-Argument>)

Effekt: Gibt die derzeitige Cursorposition am Bildschirm oder Drucker aus.

Erklärung:

Die am weitesten links liegende Cursorposition ist Null. POS gibt nicht unbedingt die physikalische Position des Druckkopfes an.

Lesen Sie hierzu auch unter LPOS nach.

Beispiel:

```
Ok 40 X = POS(0)
Ok 50 PRINT "DRUCKKOPF SITZT IN SPALTE: "; X
Ok 60 IF WIDTH.LINE ‹POS(0) THEN WIDTH.CHR = X
```

Syntax: PRINT [<Ausdruck> <,oder ;><Ausdruck [<, oder ;>]]

Effekt: Druckt Daten im Ausgabefenster aus.

Erklärung:

PRINT übermittelt Ausdrücke an das Ausgabefenster. Sie können eine beliebige Anzahl von Ausdrücken zusammen mit der PRINT-Anweisung verwenden. Die einzelnen Ausdrücke müssen durch Kommata oder Strichpunkte voneinander abgetrennt werden.

Die Interpunktionszeichen, die für die Abtrennung der einzelnen Ausdrücke verwendet werden, legen die Positionen der Ausdrücke auf dem Bildschirm fest. ST BASIC unterteilt eine Zeile in einzelne Druckzonen, die jeweils 14 Stellen umfassen. Wenn Sie die Ausdrücke in der PRINT-Anweisung durch Kommata abtrennen, wird jeder Ausdruck von ST BASIC in der nächsten verfügbaren Druckzone dargestellt. Bei Verwendung eines Strichpunktes werden die String-Ausdrücke von ST BASIC konsekutiv und ohne abtrennende Leerstellen ausgedruckt. Numerische Ausdrücke werden zusammenhängend gedruckt, wobei zwischen dem Vorzeichen und der Zahl eine Leerstelle gesetzt wird.

Wenn Sie eine Auflistung von Ausdrücken mit einem Komma abschließen, springt ST BASIC in die nachfolgende Druckzone, geht jedoch nicht weiter zu einer neuen Zeile. Schließen Sie eine Auflistung mit einem Strichpunkt ab, beläßt ST BASIC den Cursor am Ende des letzten Ausdruckes.

Anstelle von PRINT kann in ST BASIC Programmen auch ein Fragezeichen verwendet werden. ? A entspricht PRINT A.

Beispiel:

```
Ok 10 PRINT "TEST VON ST BASIC"
Ok 20 PRINT
Ok 30 A$ = "EINS" : B$ = "ZWEI" : C$ = "DREI"
Ok 40 A=23:B0567:C=5
Ok 50 PRINT A$,B$,C$
Ok 60 PRINT A$;B$;C$
Ok 70 PRINT A,B,C
Ok 80 PRINT A;B;C;
Ok 90 END
Ok RUN
TEST VON ST BASIC

EINS           ZWEI           DREI
EINSZWEIDREI
 23             567            5
 23  567  5
Ok
```

Syntax: PRINT# <Dateinummer>,<Ausdruck>,<Ausdruck>

Effekt: Gibt Daten an eine Diskettendatei aus.

Erklärung:

Die PRINT#-Anweisung schreibt Ausdrücke in die Datei, die durch die Dateinummer vorgegeben wurde. Die Dateinummer ist die Kennzahl, die Sie der Datei beim Öffnen zugewiesen hatten. Jede PRINT#-Anweisung erstellt einen eigenen Datensatz. Jeder Ausdruck in einer PRINT#-Anweisung erstellt ein eigenes Feld.

Sie können eine beliebige Anzahl von Ausdrücken zusammen mit der PRINT#-Anweisung verwenden. Die Ausdrücke müssen durch Kommata oder Strichpunkte voneinander abgetrennt werden.

PRINT# schreibt die Daten exakt in der Form in die Datei, in der sie auch mit der PRINT-Anweisung auf dem Bildschirm dargestellt würden.

Sie müssen über die entsprechenden Interpunktionszeichen genau angeben, wie die Daten in der Datei erscheinen sollen.

Beispiel:

```
1 X$ = "Lewis"
Z$ = C. S."
```

Gewünschte Darstellungsform auf der Diskette: Lewis, C.S.

Da weder vor "Lewis", noch hinter "C.S." ein Komma gesetzt wurde, schreibt die Anweisung

```
1 Ok PRINT#1,X$;Z$
Lewis C.S.
```

auf Diskette.

Wollen Sie ein Komma als Abgrenzungszeichen verwenden, müssen Sie die Anweisung

```
Ok PRINT#1,X$;",";Z$
```

verwenden, wobei das Kommazeichen als literaler String in Anführungszeichen gesetzt werden muß.

Beispiel:

```
Ok 50 PRINT#VIER.TEX; A$,B$,C$
```

Syntax: PRINT USING<String-Ausdruck>;<"Liste der Ausdrücke">;
PRINT#<Dateinummer>,USING<"String-Ausdruck">;
<Liste der Ausdrücke>

Effekt: Druckt eine Ausgabe gemäß dem vorgegebenen Format.

Erklärung:

Die PRINT USING-Anweisung druckt Daten auf den Bildschirm. Die PRINT# USING-Anweisung druckt Daten in eine Diskettendatei. Sie können Strings oder Zahlen über beliebige Anweisungen drucken. Bei der PRINT# USING-Anweisung ist die Dateinummer die Kennzahl, die Sie der Datei beim Öffnen zugewiesen hatten.

Bei beiden Anweisungen entspricht der String-Ausdruck in Anführungszeichen einer Liste von Zeichen, über die die Felder und Formate der gedruckten Daten festgelegt werden. Die Auflistung enthält die zu druckenden Begriffe, die durch Kommata oder Strichpunkte voneinander abgetrennt sein müssen. Wird die Auflistung mit einem Strichpunkt beendet, wird der Cursor am Ende des letzten Ausdruckes belassen.

Die Zeichen in der Format-Spezifikation werden durch die Daten in der Druckliste ersetzt, es sei denn, es handelt sich um literale Zeichen.

Die nachfolgende Tabelle enthält die Formatierzeichen von ST BASIC.

Formatierzeichen für String-Felder

Zeichen	Erklärung
!	Gibt der Anweisung an, daß das erste Zeichen jedes spezifizierten Strings gedruckt werden soll.
\Zeichen\	Zeichen plus 2 zeigt die Gesamtanzahl von Zeichen an, die von dem spezifizierten String gedruckt werden soll.
&	Spezifiziert ein String-Feld mit variabler Länge.

Formatierzeichen für numerische Felder

Zeichen	Erklärung
#	Repräsentiert jede Ziffernposition in einem numerischen Feld.
.	Fügt Nullen ein, um eventuelle Ziffernpositionen aufzufüllen.
+	Druckt das Vorzeichen der Zahl, Plus oder Minus, vor der gedruckten Zahl aus.
–	Druckt negative Zahlen mit einem vorangestellten Minuszeichen aus.
**	Füllt Leerstellen in einem numerischen Feld mit Sternchen auf.
$$	Setzt direkt links neben die gedruckte Zahl ein Dollarzeichen.
**$	Füllt Leerstellen mit Sternchen auf und setzt links neben die Zahl ein Dollarzeichen.
,	Fügt links neben dem Dezimalpunkt nach jeder dritten Ziffer ein Komma ein.
^^^^	Spezifiziert ein Exponential-Format.
_	Druckt das nachfolgende Zeichen als literales Zeichen.

Sie können String-Konstanten in einen Format-String einfügen, wie im nachfolgenden Beispiel demonstriert wird:

Beispiel:

```
Ok 10 PRINT USING "DAS IST DATEI _###";4
Ok RUN
DAS IST DATEI # 4
Ok
```

PUT **PUT #1,5** **ANWEISUNG**

Syntax: PUT[#]<Dateinummer>,<Datensatz-Nummer>

Effekt: Schreibt einen Datensatz von einem Speicher in eine Random-Diskettendatei.

Erklärung:

Die Dateinummer ist die Kennzahl, die Sie der Datei beim Öffnen zugewiesen hatten. Die Datensatz-Nummer ist optional verwendbar. Sofern Sie eine Datensatz-Nummer angeben, muß diese bei Eins beginnen und in sequentieller Folge fortfahren. Für die Zuordnung von Datensatz-Nummern in einer Datei wird am besten eine FOR TO NEXT-Schleife geschrieben. Geben Sie keine Datensatz-Nummer an, verwendet PUT die nächste Datensatz-Nummer nach der letzten GET- oder PUT-Anweisung. Die größte gültige Datensatz-Nummer ist 32767.

Sie sollten vor der Verwendung von PUT LSET oder RSET setzen, um die Daten in den Random-Speicher zu bringen.

Beispiel:

```
Ok 100 LSET Q$=X$
Ok 120 PUT#2,RCORD%
```

QUIT **QUIT** **BEFEHL**

Syntax: QUIT

Effekt: Beendet ST BASIC und kehrt zurück zum GEM.

Erklärung:

QUIT schließt alle Dateien und bringt Sie zum GEM-Befehlslevel zurück. Jedes im Arbeitsspeicher befindliche Programm wird gelöscht, sofern es nicht zuvor gespeichert wurde.

QUIT entspricht SYSTEM.

Beispiel:

```
Ok QUIT
```

Syntax: RANDOMIZE [<numerischer Ausdruck>]

Effekt: Setzt den Random-Zahlengenerator.

Erklärung:

RANDOMIZE wird zusammen mit der RND-Funktion verwendet, um zufällig bestimmte Zahlen zu generieren. Wenn Sie den optional verwendbaren numerischen Ausdruck nicht angeben, fragt ST BASIC nach einer Zahl, auf der RANDOMIZE basieren soll.

Falls Sie am Anfang eines Programmes, das mit zufällig bestimmten Zahlen arbeitet, kein RANDOMIZE mit dem Parameter Null verwenden, gibt die Funktion RND bei jedem Programmlauf dieselbe Sequenz von Zahlen in gleicher Reihenfolge aus.

Lesen Sie auch unter RND nach, um weitere Informationen über die Generierung zufällig bestimmter Zahlen zu erhalten.

Beispiel:

```
Ok 10 RANDOMIZE 0
Ok 20 FOR X = 1 TO 10
Ok 30 PRINT RND
Ok 40 NEXT X
Ok RUN
 .957395
 .427143
 .806267
 .0206223
 .86628
 .886706
 .435054
 .199773
 .505868
 .801594
Ok
```

Syntax: READ<Variable>, <Variable>

Effekt: Ordnet Werte aus einer DATA-Anweisung Variablen zu.
Erklärung:

Die Anweisungen READ und DATA werden immer in Kombination verwendet. READ ordnet die in DATA aufgelisteten Werte nacheinander einer damit korrespondierenden Auflistung von Variablen zu. Die Variablen können numerische Ausdrücke oder Strings sein. Sie müssen vom Typ her den Konstanten-Werten in der DATA-Anweisung entsprechen. Andernfalls würde ein Fehler auftreten.

Sie können eine READ-Anweisung mit mehreren DATA-Anweisungen zusammen verwenden. Umgekehrt können Sie mehrere READ-Anweisungen mit einer DATA-Anweisung kombinieren. Ist die Anzahl von Werten in der DATA-Anweisung größer als die Anzahl von Variablen in der READ-Anweisung, greift die nächste READ-Anweisung die verbliebenen Konstanten aus der ersten DATA-Anweisung auf und ordnet diese den Variablen ihrer Liste zu. Gibt es keine nachfolgende READ-Anweisung, werden die verbliebenen Daten ignoriert.

Gibt es weniger Werte in der DATA-Anweisung als in der READ-Anweisung, wird die nächste DATA-Anweisung angesteuert und ausgelesen. Folgt keine weitere DATA-Anweisung, tritt ein Fehler Nr. 4 (zuwenig Werte) auf.

Sie können über die RESTORE-Anweisung DATA-Begriffe vom Anfang einer spezifizierten Zeilennummer neu lesen lassen.

Lesen Sie hierzu auch unter DATA und RESTORE nach.

Beispiel:

```
Ok 10 READ X,Y,Z
Ok 20 RESTORE
Ok 30 DURCHSCHNITT = (X+Y+Z)/3
Ok 40 DATA 23.4,89.2,77
Ok 50 PRINT DURCHSCHNITT
Ok 60 READ X,Y,Z
Ok 70 ERGEBNIS = X*Y*Z
Ok 80 PRINT ERGEBNIS
Ok 90 END
Ok RUN
  63.2
  160720
Ok
```

Syntax: REM <Anmerkung>

Effekt: Ermöglicht Anmerkungen im Programmcode.

Erklärung:

Anmerkungen dienen dazu, die Logik eines Programmes deutlich zu machen. REMs erscheinen im Programmlisting in der Form, in der Sie sie geschrieben hatten, haben jedoch keine Auswirkungen auf die Programmausführung. Anmerkungen dürfen maximal 245 Zeichen lang sein. Schreiben Sie eine Anmerkung, die länger ist als die Bildschirmbreite, können Sie die Zeile mit einem Zeilenvorschub verlängern.

Wenn Sie über eine GOTO- oder GOSUB-Anweisung in eine REM-Zeile springen, wird der Programmlauf in der ersten ausführbaren Zeile nach REM fortgeführt.

Das einfache Apostroph-Zeichen bewirkt denselben Effekt wie REM. So ist beispielsweise

```
Ok 100 'das ist eine Anmerkung
```

eine gültige Anweisung.

Beispiel:

```
Ok 10 REM DIESES PROGRAMM ERRECHNET QUADRATZAHLEN
Ok 20 INPUT "ZAHL EINGEBEN, DIE POTENTIERT WERDEN SOLL";X
Ok 30 S=X*X
Ok 40 PRINT S
Ok 50 'RÜCKKEHR ZUR ZEILE FÜR DIE ZAHLENEINGABE
Ok 60 GOTO 20
Ok 70 END
Ok RUN
```

Syntax: 9 RENUM [<neue erste Zeile>][,<Anfangszeile>][,<Erhöhung>]

Effekt: Numeriert Programmzeilen neu.

Erklärung:

Enthält Ihr Programm unregelmäßige Zeilennummern, die durch nachträgliches Einfügen neuer Zeilen zwischen bereits erstellte Zeilen entstanden sind, können Sie das gesamte Programm neu durchnumerieren, ohne dabei GOTO- oder andere Adressenabhängige Anweisungen anpassen zu müssen.

Wird RENUM alleine verwendet, wird das Programm in Zehnerschritten, beginnend bei Zeilennummer 10, fortlaufend durchnumeriert.

Sie können auch eine neue Zeilennummer für die erste Zeile des Programmes vorgeben. Zudem ist möglich, eine Anfangszeilennummer anzugeben, ab der das Programm neu numeriert werden soll.

Außerdem können Sie festlegen, in welchen Schritten die Zeilennummern erhöht werden. Mit der Anweisung

RENUM 10,30,10

beginnt die Neunumerierung in der ehemaligen Zeile 30, die die neue Zeilennummer 10 erhält. Die Zeilennummern werden in Zehnerschritten erhöht.

RENUM 10,30,20

bewirkt eine Neunumerierung bei der ehemaligen Zeile 30, die die neue Zeilennummer 10 zugeordnet bekommt. Die Erhöhung erfolgt in Zwanzigerschritten, also 10, 30, 50, 70 usw.

Sie können jede einzelne Option von RENUM alleinstehend verwenden. Wenn Sie jedoch beispielsweise nur eine unterschiedliche Erhöhung festlegen, sollten Sie für die beiden ersten Optionen Kommata als Stellenmerker setzen, um kennzuzeichnen, daß Sie einen Erhöhungswert und nicht eine neue erste Zeilennummer oder Anfangszeile angeben wollen (z.B. RENUM ,,20).

RENUM paßt alle Zeilennummern-Referenzen in GOTO-, GOSUB-, IF ... THEN ... ELSE-, ON ... GOTO- und ON ... GOSUB-Anweisungen an die neue Zeilennumerierung an. Haben Sie in einer dieser Anweisungen eine nicht existierende Zeilennummer verwendet, wird diese unverändert belassen.

Sie können RENUM nicht dazu verwenden, um die Reihenfolge der Programmzeilen zu verändern.

RENUM legt eine Datei mit der Bezeichnung BASIC.WRK auf der derzeit im Laufwerk befindlichen Diskette an. Aus diesem Grund darf die eingelegte Diskette nicht mit einem Schreibschutz versehen sein.

Beispiel:

```
Ok 15 X=5
Ok 20 Z=3
Ok 25 Y=10
Ok 30 PRINT X+Y-Z
Ok RENUM
    LIST
10 X=5
20 Z=3
30 Y=10
40 PRINT X+Y-Z
Ok
```

Syntax: REPLACE [<Dateiname>][,<Zeilennummern-Liste>]

Effekt: Ersetzt eine alte Version einer Datei durch eine neue Version.

Erklärung:

REPLACE wird zusammen mit OLD oder LOAD verwendet. Nachdem Sie eine alte Datei geladen und überarbeitet haben, wird mit REPLACE die überarbeitete Fassung der Datei auf Diskette abgelegt, wobei die alte Datei gelöscht wird.

Wenn Sie einen Dateinamen angeben, speichert REPLACE das Originalprogramm unter <Dateiname> und nicht unter dem Original-Dateinamen. Sie können Teile eines Programmes speichern, wenn Sie die betreffenden Zeilennummern hinter REPLACE angeben.

In der Regel entspricht REPLACE dem Befehl SAVE. Es besteht lediglich der Unterschied, daß mit REPLACE der Name der Datei, die Sie speichern wollen, bereits einer anderen Datei zugeordnet worden sein darf. In untenstehendem Beispiel wird das Programm ZAEHLEN in den Arbeitsspeicher geladen, die Zeile 130 ersetzt und die überarbeitete Fassung auf Diskette gespeichert.

Beispiel:

```
Ok OLD ZAEHLEN
Ok 130 IF X = 10 THEN END
Ok REPLACE
Ok
```

Syntax: RESET

Effekt: RESET legt den Inhalt des Ausgabefensters in den Grafik-Speicher ab.

Erklärung:

Wenn die Option "Buffered Graphics" aktiviert ist, kopiert RESET den Inhalt des Ausgabefensters in den Grafik-Speicher. Dadurch kann eine Grafik auf Diskette abgelegt, und nach der Ausführung weiterer Grafik-Operationen wieder ins Ausgabefenster eingebracht werden. Über die Anweisung OPENW wird der Inhalt des Grafik-Speichers wieder zurück ins Ausgabefenster gebracht.

Beispiel:

```
10 COLOR 1,1,1,1,1:FULLW 2
Ok 20 CIRCLE 100,100,50
Ok 30 RESET: 'LEGT DAS BILD IM SPEICHER AB
Ok 40 CLEARW 2
Ok 50 PCIRCLE 100,100,50
Ok 60 FOR I=1 TO 1000:NEXT
Ok 70 OPENW 2
Ok 80 END
```

Syntax: RESTORE <Zeilenangabe>
Effekt: Liest DATA-Anweisungen neu.

Erklärung:

RESTORE ermöglicht eine Spezifikation der DATA-Anweisung, die Sie zusammen mit READ-Anweisungen verwenden wollen. RESTORE sucht den ersten Begriff der ersten DATA-Anweisung in oder nach der angegebenen Programmzeile und kennzeichnet diesen als Startpunkt für die nächste READ-Anweisung.

Sie können jede beliebige DATA-Anweisung durch Angeben der betreffenden Zeilennummer als Objekt einer RESTORE-Anweisung festlegen. Die mit RESTORE verwendete Zeilenangabe muß sich nicht auf die DATA-Anweisung beziehen; die angegebene Zeilennummer muß auch nicht unbedingt im Programm vorhanden sein. Die nächste READ-Anweisung sucht die DATA-Anweisung direkt in der angegebenen Programmzeile, bzw. in den darauffolgenden Zeilen.

Beispiel:

```
Ok 10 READ X,Y,Z
Ok 20 RESTORE
Ok 30 DURCHSCHNITT = (X + Y + Z)/3
Ok 40 DATA 23.4, 89.2, 77
Ok 50 PRINT  DURCHSCHNITT
Ok 60 READ X,Y,Z
Ok 70 ERGEBNIS = X * Y * Z
Ok 80 PRINT  ERGEBNIS
Ok 90 END
Ok RUN
  63.2
  160720
Ok
```

Syntax: RESUME (0)
RESUME NEXT
RESUME <Zeilenangabe>

Effekt: Fährt nach einem Fehler mit dem Programmlauf fort.

Erklärung:

Nach der Aufdeckung und Behebung eines Fehlers wird mit RESUME der normale Programmlauf wieder aufgenommen. Sie dürfen eine RESUME-Anweisung nur am Ende einer Fehleraufdeckungs-Routine schreiben. Die Ausführung einer RESUME-Anweisung an einer anderen Stelle innerhalb eines Programmes würde einen unauffindbaren Fehler ergeben.

RESUME allein oder mit einer nachgestellten Null gibt die Programmkontrolle an die Anweisung zurück, bei der der Fehler auftrat.

RESUME NEXT übergibt die Programmkontrolle an die nächste Anweisung nach der Anweisung, die den Fehler verursachte.

RESUME <Zeilenangabe> übergibt die Programmkontrolle an die angegebene Zeilennummer.

Beispiel:

```
Ok 100 ON  ERROR  GOTO 700
 .
 .
 .
Ok 700 IF (ERR = 300) AND (ERR = 150) THEN PRINT
"MINDESTANZAHL ABHÄNGIGER WERTE IST 1": RESUME 140
```

Syntax: RETURN

Effekt: Übergibt die Kontrolle von einer Unterroutine an die Anweisung, die dem letzten GOSUB folgte.

Erklärung:

RETURN übergibt die Programmausführung an die erste ausführbare Anweisung im Hauptprogramm hinter einem Unterroutinen-Aufruf. Die Unterroutine kann eine GOSUB- oder eine ON ... GOSUB-Anweisung sein.

Beispiel:

```
Ok 10 GOSUB ALPHA
Ok 20 REM RÜCKKEHRPUNKT DER UNTERROUTINE
Ok 30 PRINT A
Ok 40 GOTO 200
Ok ALPHA: REM BEGINN DER UNTERROUTINE
Ok 110 A=5*6
Ok 120 RETURN
Ok 200 END
Ok RUN
 30
Ok
```

Syntax: X$ = RIGHT$(<Zielstring>,<Anzahl der Zeichen>)

Effekt: Gibt die letzten Zeichen, gerechnet von rechts, einer Zeichenkette aus.

Erklärung:

RIGHT$ ordnet die von Ihnen angegebene Anzahl von Zeichen in einem Zielstring, gerechnet von rechts, einer neuen String- Variablen zu. Ist die angegebene Zeichenanzahl größer oder gleich der Stringlänge, wird der gesamte String ausgegeben. Geben Sie als Anzahl Null an, wird ein Null-String ausgegeben.

Beispiele:

```
Ok 10 A$ = "Marketing-Strategie"
Ok 20 B$ = "Regionale Aktionen"
Ok 30 C$ = "Testergebnisse"
Ok 40 INPUT "KATALOG NUMMER"; KATALOG$
Ok 50 IF RIGHT$(KATALOG$,1) = "1" THEN PRINT "SIE HABEN
GEWAEHLT: "
Ok 60 PRINT "TEST KATALOG SERIE 1"
Ok 70 PRINT "BITTE WAEHLEN SIE AUS: "
Ok 80 PRINT A$
Ok 90 PRINT B$
Ok 100 PRINT C$
Ok RUN
KATALOG NUMMER? ATARI GESAMTKATALOG 201
SIE HABEN GEWAEHLT:
TEST KATALOG SERIE 1.
BITTE WAEHLEN SIE AUS:
Marketing-Strategie
Regionale Aktionen
Testergebnisse
Ok
```

```
Ok 10 A$ = "ST BASIC"
Ok 20 B$ = RIGHT$(A$,5)
Ok 30 PRINT B$
RUN
BASIC
Ok
```

RND **X = RND** **FUNKTION**
X = RND(Y)
X = RND(0)
X = RND (-Y)

Syntax: X = RND [(<numerischer Ausdruck>)]

Effekt: Generiert eine Random-Zahl und gibt sie aus.

Erklärung:

RND gibt eine uniform verteilte Zufallszahl, die zwischen 0 und 1 liegt, aus. Falls Sie vor Verwendung der RND-Anweisung keine RANDOMIZE-Anweisung gesetzt haben, wird dieselbe Folge zufällig gewählter Zahlen bei jedem Programmlauf wieder generiert.

Die Funktionsweise von RND ist unterschiedlich, abhängig davon, ob der numerische Ausdruck eine positive oder negative Zahl oder eine Null ist:

RND (<positive Zahl>) gibt die nächste Zahl der derzeitigen Sequenz aus.

RND (0) gibt die zuletzt erstellte Random-Zahl aus, ohne die derzeitige Sequenz zu beeinflussen.

RND (<negative Zahl>) setzt den Random-Zahlengenerator mit der negativen Zahl neu und gibt die erste zufällig gewählte Zahl innerhalb der neuen Sequenz aus.

Die Angabe des numerischen Ausdruckes ist optional. Wird hierfür keine Angabe gemacht, handelt RND, als hätten Sie einen positiven Ausdruck als Argument eingesetzt.

Anmerkung: Lesen Sie hierzu auch unter RANDOMIZE nach.

Beispiel:

```
Ok 10 RANDOMIZE
Ok 20 X = RND
Ok 30 WURF$ = "ZAHL"
Ok 40 IF X >.5 THEN WURF$ = "KOPF"
Ok 50 INPUT "KOPF ODER ZAHL";W$
Ok 60 IF W$ = WURF$ THEN PRINT "GEWONNEN" ELSE PRINT
"VERLOREN"
Ok RUN

Angabe der Random-Zahl (zwischen -32768 und +32767)? 2
KOPF ODER ZAHL? ZAHL
GEWONNEN
Ok
```

Syntax: RSET <String-Variable>=<String-Ausdruck>

Effekt: Verschiebt eine Zeichenkette in eine angegebene String-Variable, ohne die String-Variable neu zuzuordnen.

Erklärung:

RSET wird in der Regel dazu verwendet, um Daten in Datei-Speicher einzubringen. Dazu werden die Daten in Variablen zurückgesetzt, die über eine vorhergehende FIELD-Anweisung in Dateispeichern abgelegt wurden.

Ist der zu verschiebende String kürzer als der Ziel-String, setzt RSET den String rechtsbündig an und füllt ihn links mit Leerzeichen auf. Ist die Zeichenkette länger als der Ziel-String, werden die zusätzlichen Zeichen von RSET nicht beachtet.

Zahlen müssen mit RSET oder LSET umgewandelt werden, bevor sie mit MKS$, MKI$ oder MKD$ verwendet werden können.

Beispiel:

```
Ok 10 OPEN "R",#3,"TEST"
Ok 20 FIELD #3,20 AS A$,20 AS B$
Ok 30 RSET A$=X$
Ok 40 RSET B$=STRESS$
```

RUN
RUN ,200
RUN DATEI.BAS

Syntax: RUN
RUN <,Zeilenangabe>
RUN <Dateiname>

Effekt: Beginnt einen Programmlauf.

Erklärung:

RUN führt ein derzeit im Speicher oder auf einer Diskettendatei befindliches Programm aus. Die Programmausführung beginnt in der ersten Programmzeile, sofern Sie nicht andere Vorgaben gemacht haben. Ist das auszuführende Programm in einer Diskettendatei abgelegt, löscht RUN jedes im Arbeitsspeicher befindliche Programm, bevor das angegebene Programm geladen wird.

Programmausgaben erscheinen im Ausgabefenster.

Um den Programmlauf anzuhalten und den BREAK-Modus zu aktivieren, betätigen Sie die Tastenkombination [CONTROL][G] oder klicken auf die Option "Break" im Menü "Run".

Wollen Sie mit der Programmausführung fortfahren, geben Sie CONT ein oder drükken Sie [RETURN].

Wenn Sie den BREAK-Modus verlassen und gleichzeitig die Programmausführung beenden wollen, geben Sie STOP oder END ein. Mit [CONTROL][C] wird der Programmlauf abgebrochen und Sie kehren zurück zu ST BASIC.

SAVE **SAVE DATEI** **BEFEHL**
SAVE DATEI,20-30
SAVE DATEI,10,30,70,80
SAVE DATEI,-30

Syntax: SAVE [<Dateiname>], [<Zeilenangabe-Liste>]

Effekt: Speichert Programmzeilen auf Diskette.

Erklärung:

SAVE legt ein Programm, bzw. die von Ihnen angegebenen Zeilen eines Programmes in einer Diskettendatei ab. SAVE ergänzt den Dateinamen mit dem Extender .BAS, sofern Sie keine anderen Vorgaben machen. Wenn Sie versuchen, ein Programm mit SAVE unter einem bereits auf der Diskette befindlichen Dateinamen zu speichern, erhalten Sie eine Fehlermeldung. SAVE ersetzt eine bereits bestehende Diskettendatei nicht durch ein neues Programm.

Um eine bestehende Diskettendatei mit einem neuen Programm zu überschreiben, verwenden Sie REPLACE.

SGN **X = SGN(Y)** **FUNKTION**

Syntax: X = SGN(<numerischer Ausdruck>)

Effekt: Gibt das Vorzeichen einer Zahl aus.

Erklärung:

SGN gibt 1 aus, wenn der numerische Ausdruck positiv ist, −1, wenn er negativ ist, und 0, wenn der Ausdruck Null entspricht.

Beispiel:

```
Ok 10 X = SGN(-3)
Ok 20 Y = SGN(0)
Ok 30 Z = SGN(2)
Ok 40 PRINT X
Ok 50 PRINT Y
Ok 60 PRINT Z
Ok RUN
−1
 0
 1
Ok
```

Syntax: X = SIN(<numerischer Ausdruck>)

Effekt: Gibt den Sinus eines Argumentes, das in Radian ausgedrückt wird, aus.

Erklärung:

Die Funktion SIN geht davon aus, daß der Ausdruck ein in Radian angegebener Winkel ist. Um Gradzahlen in Radianzahlen umzuwandeln, multiplizieren Sie die Gradzahl mit Pi und teilen das Ergebnis durch 180 (Pi = 3.141593). SIN wandelt Integerzahlen in Realzahlen um und gibt Realzahlen aus.

Beispiel:

```
Ok 10 PRINT SIN(23)
Ok RUN
 -.84622
Ok
```

SOUND **SOUND STIMME, LAUTSTÄRKE, NOTE, OKTAVE, STIMME** **ANWEISUNG**

Syntax: SOUND <numerischer Ausdruck>, <numerischer Ausdruck>, <numerischer Ausdruck>, <numerischer Ausdruck>, <numerischer Ausdruck>,

Effekt: SOUND steuert die drei Tonkanäle.

Erklärung:

Mit SOUND werden Musiknoten generiert.

Für STIMME (voice) wird die Kennzahl des verwendeten Tonkanales (1-3) angegeben.

Über LAUTSTÄRKE (volume) kann die Lautstärke geregelt werden (0 = Aus, 15 = größte Lautstärke).

Mit NOTE (note) und OKTAVE (octave) wird die Tonhöhe einer Note eingestellt. Sie geben eine Oktaven-Kennzahl (zwischen 1 und 8), sowie eine Noten-Kennzahl (zwischen 1 und 12) an. Die Noten-Kennzahlen entsprechen den Noten-Positionen auf der Tonleiter. Ein 440 Hz A entspricht Note 10 in Oktave 4.

DAUER (duration) entspricht der Zeitdauer (gerechnet in 1/50 Sekunden), über die eine Note gehalten wird, bevor der nächste Ton beginnen soll. Die letzte SOUND-Anweisung für jede Stimme sollte grundsätzlich den Ton abschalten (z.B. SOUND 3,0,0,0,0). Sie können die SOUND-Anweisung auch als Timing-Funktion verwenden. Setzen Sie hierzu die Lautstärke auf 0 und die Dauer auf die gewünschte Verzögerung.

Beispiel:

```
Ok 10 SOUND 1,8,12,4,25
Ok 20 SOUND 1,8,9,4,25
Ok 30 SOUND 1,0,0,0,0,
```

Syntax: X$ = SPACE$(<numerischer Ausdruck>)

Effekt: Gibt einen String mit Leerzeichen aus.

Erklärung:

SPACE$ gibt die über den numerischen Ausdruck von Ihnen angegebene Anzahl von Leerzeichen aus. Der Wert des Ausdruckes muß zwischen 0 und 255 liegen.

Anmerkung: Wenn Sie lediglich zum Drucken eine Anzahl von Leerzeichen erstellen wollen, sollten Sie besser die Funktion SPC (X) verwenden.

Beispiel:

```
Ok 10 X = 10
Ok 20 FOR V = 1 TO 5
Ok 30 PRINT SPACE$(X); ""
Ok 40 NEXT V
Ok 50 FOR Z = 1 TO 21
Ok 60 PRINT "-";
Ok 70 NEXT Z
Ok RUN
```

SPC **PRINT SPC(X)** **FUNKTION**

Syntax: PRINT SPC(<numerischer Ausdruck>)

Effekt: Gibt Leerzeichzen an eine PRINT-Anweisung aus.

Erklärung:

SPC druckt die Anzahl von Leerzeichen aus, die Sie über den numerischen Ausdruck vorgeben. Der Ausdruck muß zwischen -32768 und 32767 liegen.
Ist die vorgegebene Anzahl von Leerzeichen größer als die eingestellte Zeilenbreite für den Drucker, wird der Wert über MOD entsprechend umgewandelt. (Nähere Informationen über die Umwandlung von Zeichen mit MOD erhalten Sie unter CHR$.)

Wenn beispielsweise die Zeilenbreite auf 72 Zeichen eingestellt ist und der numerische Ausdruck mit 100 eingegeben wird, fügt SPC 28 Leerzeichen ein.

Ist der numerische Ausdruck größer als 255, entspricht die Anzahl eingefügter Leerzeichen dem numerischen Ausdruck MOD 255.

Anmerkung: Verwenden sie SPC immer nur zusammen mit PRINT, LPRINT und PRINT#.

Beispiel:

```
Ok 10 PRINT "ALPHABET"
Ok 20 PRINT
Ok 30 PRINT "A"SPC(3)"a"SPC(7)"B"SPC(3)"b"SPC(7)"C"SPC(3)"c"
Ok RUN
ALPHABET
A  a          B  b          C  c
```

SQR **X = SQR(Y)** **FUNKTION**

Syntax: X = SQR(numerischer Ausdruck)

Effekt: Gibt die Quadratwurzel einer Zahl aus.

Erklärung:

Die Zahl darf kein negatives Vorzeichen haben. SQR gibt eine Realzahl aus.

Beispiel:

```
Ok 10 PRINT SQR(9)
Ok RUN
  3
Ok
```

Syntax: STEP
STEP <,Zeilenangabe>
STEP <Dateiname>

Effekt: Führt ein Programm zeilenweise aus.

Erklärung:

STEP führt ein Programm zeilenweise aus, wobei jede Zeile und eventuelle Ausgabe dargestellt wird. Erst nach Betätigen von [RETURN] wird die nächste Zeile ausgeführt.

Um den STEP-Modus zu verlassen, geben Sie CONT ein. Damit wird der Programmlauf weitergeführt. Wollen Sie die Programmausführung nach STEP abbrechen, geben Sie END ein.

Beispiel:

```
Ok 10 X = 9
Ok 20 PRINT X
Ok 30 PRINT "WIE GEHT ES DIR?"
Ok 40 END
Ok STEP, 10
S 10 X=9
BR [RETURN]
S 20 PRINT X
BR [RETURN]
S 30 PRINT "WIE GEHT ES DIR?"
BR [RETURN]
WIE GEHT ES DIR?
S 40 END
BR [RETURN]
Ok
```

Syntax: STOP

Effekt: STOP hält die Programmausführung an und übergibt die Kontrolle über BASIC an das Befehlsfenster.

Erklärung:

Nach der Eingabe von STOP ist das Programm im BREAK-Level. Sie können ein Programm an jeder Stelle über STOP anhalten. Im Gegensatz zu END beläßt STOP Dateien geöffnet, springt in den BREAK-Modus und ermöglicht ein Fortfahren mit der Programmausführung zu einem späteren Zeitpunkt. Zudem wird bei Verwendung von STOP die Meldung "STOP" ausgegeben.

Mit CONT oder [RETURN] können Sie den Programmlauf wieder aufnehmen.

Beispiel:

```
Ok 10 A=4:B=6:C=8
Ok 20 PRINT A,A*B
Ok 30 STOP
Ok 40 PRINT C*A
Ok 50 END
Ok RUN
  4           24
Stop at line 30
Br CONT
  32
Ok
```

Syntax: X$ = STR$(<numerischer Ausdruck>)

Effekt: Gibt einen String aus, der das dem Argument entsprechende Dezimalzeichen enthält.

Erklärung:

Der ausgegebene String enthält die Standard-Repräsentation des Ausdruckes. Er beinhaltet die Zeichen, die ausgedruckt würden, wenn eine PRINT-Anweisung ausgeführt worden wäre.

Bei positiven Zahlen setzt STR$ ein Leerzeichen für das Pluszeichen vor die Zahl. STR$ löscht alle Leerzeichen, die hinter einer Zahl angegeben wurden.

VAL ist das Gegenstück zu STR$.

Lesen Sie hierzu auch unter OCT$ und HEX$ nach.

Beispiel:

```
Ok 10 VORWAHL = 089
Ok 20 PRINT STR$(VORWAHL) + " (MUENCHEN)"
Ok RUN
  089 (MUENCHEN)
Ok
```

STRING$** **X$ = STRING$(Y,A$) **FUNKTION**
X$ = STRING$(Y,N)

Syntax: X$ = STRING$(<numerischer Ausdruck>,<numerischer oder String-Ausdruck>)

Effekt: Gibt einen String mit der angegebenen Länge aus. Die Zeichen werden durch das zweite Argument definiert.

Erklärung:

Der erste numerische Ausdruck gibt die Länge des Strings aus, der über STRING$ ausgegeben wird. Er muß zwischen 0 und 255 liegen.

Für den zweiten Parameter können Sie sowohl einen numerischen, als auch einen String-Ausdruck verwenden. Innerhalb eines numerischen Ausdruckes muß die Angabe eines Zeichens über seinen ASCII-Code erfolgen. Ein String-Zeichen kann in beliebiger Weise angegeben werden.

STRING$ gibt eine Zeichenkette in der angegebenen Länge aus, die das Zeichen enthält, das entweder über den ASCII-Code oder den ersten Buchstaben des String-Ausdruckes spezifiziert wurde.

STRING$ produziert eine geringfügigere Speicheraufsplittung und arbeitet erheblich schneller als ein Verketten. Wenn Sie eine Zeichenkette erstellen, die eine Anzahl verschiedener Zeichen enthält, ist die Verwendung von STRING$ oder SPACE$ für die Generierung eines Strings in der erforderlichen Länge, sowie die Verwendung von MID$ für das Einbringen individueller Zeichen in diesen String effizienter als ein Verketten von Zeichenketten.

Beispiel:

```
Ok 10 Z$ = STRING$(20,"*")
Ok 20 PRINT Z$
Ok RUN
********************
Ok
```

Syntax: SWAP <erste Variable>,<zweite Variable>

Effekt: Vertauscht die Werte zweier Variablen.

Erklärung:

Sie können jeden Variablentyp mit SWAP austauschen. Allerdings müssen die beiden Variablen, zwischen denen Werte getauscht werden sollen, vom gleichen Typ sein. Es besteht die Möglichkeit, Array-Variablen auszutauschen. Arrays selbst können dagegen nicht getauscht werden.

```
SWAP A%(3),B%(7,5)     ist eine gültige Anweisung
SWAP A%(),B%()         ist ungültig
```

Beispiel:

```
Ok 10 X$ = "THOMAS BERGER"
Ok 20 Y$ = "ANNE MEIER"
Ok 30 O$ = "EHEMALIGER"
Ok 40 C$ = "NEUER"
Ok 50 M$ = " MARKETING MANAGER: "
Ok 60 PRINT O$;M$;X$
Ok 70 SWAP X$,Y$
Ok 80 SWAP O$,C$
Ok 90 PRINT O$;M$;X$
Ok RUN
EHEMALIGER MARKETING MANAGER: THOMAS BERGER
NEUER MARKETING MANAGER: ANNA MEIER
Ok
```

SYSTAB **X = PEEK(SYSTAB+OFFSET)** **VARIABLE**

Syntax: X = PEEK(SYSTAB+OFFSET)

Effekt: Systemzeiger-Tabelle.

Erklärung:

SYSTAB ist die Anfangs-Speicheradresse einer Tabelle mit Systemparametern und -zeigern. Mit Ausnahme der Adresse SYSTAB+2, die eine READ/WRITE-Speicherstelle ist, ist SYSTAB eine READ/ONLY-Speicherstelle.

SYSTAB enthält lediglich 2-Byte Werte. Lediglich SYSTAB+20, der Grafikspeicher-Zeiger, enthält eine 4-Byte Long-Integeradresse.

Der Grafikspeicher umfaßt 32768 Bytes. SYSTAB ist folgendermaßen organisiert:

Offset	Funktion
0	Grafik-Auflösung (Ebenen) 1 = HI, 2 = MED, 4 = LO
2	Aussehen der "Ghostline" im Editor (siehe nachstehende Tabelle)
*4	EDIT AES Handhabung
*6	LIST AES Handhabung
*8	OUTPUT AES Handhabung
*10	COMMAND AES Handhabung
12	EDIT Öffnungs-Flag (0 = geschlossen, 1 = geöffnet)
14	LIST Öffnungs-Flag (0 = geschlossen, 1 = geöffnet)
16	OUTPUT Öffnungs-Flag (0 = geschlossen, 1 = geöffnet)
18	COMMAND Öffnungs-Flag (0 = geschlossen. 1 = geöffnet)
20	Grafik-Speicher (4-Byte Zeiger auf 32768 Byte Speicher, sofern BUFFERED GRAPHICS aktiviert ist)
**24	GEMFLAG (0 = Normal, 1 = Aus)

BIT	BESCHREIBUNG
0	Verstärkt
1	Intensität
2	Schräg
3	Unterstrichen
4	Invertiert
5	Schattiert

* Die Verwendung der mit einem Sternchen (*) gekennzeichneten Offsets setzt Kenntnisse über das TOS Betriebssystem voraus.

** GEMDOS kann dazu verwendet werden, um die Wechselwirkung zwischen ST BASIC und GEM abzuschalten und dadurch die Verarbeitungsgeschwindigkeit zu steigern. Ist BASIC abgeschaltet, können keinerlei BASIC-Funktionen ausgeführt werden, bei denen der Bildschirm, die Maus oder die Tastatur involviert ist. Disk I/O- und Verarbeitungsfunktionen sind verfügbar. In Ihrem Programm muß diese Wechselwirkung wieder aktiviert werden, bevor Anwender-Eingaben in irgendeiner Form angenommen werden können.

SYSTEM SYSTEM BEFEHL

Syntax: SYSTEM

Effekt: Verläßt ST BASIC und kehrt zurück zum GEM.

Erklärung:

SYSTEM schließt alle Dateien und bringt Sie zurück zum GEM- Befehlslevel. Jedes im Arbeitsspeicher befindliche Programm, das nicht zuvor auf Diskette gespeichert wurde, wird damit gelöscht.

SYSTEM ist von der Funktion her identisch mit QUIT.

Beispiel:

```
Ok SYSTEM
```

Syntax: PRINT TAB(<Tabulatorposition>)

Effekt: Bewegt den Cursor an eine angegebene Tabulatorposition.

Erklärung:

TAB wird in Kombination mit PRINT, LPRINT und PRINT# verwendet.

Die Angabe der Tabulatorposition muß im Bereich zwischen -32768 und +32767 liegen. Liegt die derzeitige Druckposition bereits hinter der spezifizierten Tabulatorposition, springt TAB in die nächste Zeile und dort an die Tabulatorposition, die Sie vorgegeben haben. Spalte 1 ist die am weitesten links liegende Tabulatorposition. Die äußerste rechte Position wird durch eine WIDTH-Anweisung definiert. Ist die angegebene Position größer als 255, wird die Angabe über MOD 255 umgerechnet. Ist die Position größer oder gleich der vorgegebenen Zeilenbreite, wird ebenfalls über MOD (Breite) umgerechnet.

Nähere Informationen über das Umrechnen von Werten mit MOD erhalten Sie unter CHR$.

Beispiel:

```
Ok 10 PRINT "1985 EINKUENFTE IM QUARTAL"
Ok 20 PRINT
Ok 30 PRINT TAB (10)"WINTER"
Ok 40 PRINT TAB (70)"ZU WEIT"
Ok 50 PRINT TAB (100)"SOMMER"
Ok 60 EN
Ok RUN
1985 EIN  UENFTE IM QUARTAL
          WINTER
ZU WEI
                              SOMMER
```

Syntax: X = TAN(<Winkel in Radian>)

Effekt: Gibt die Tangente einer Zahl aus.

Erklärung:

Die Funktion TAN arbeitet auf der Basis trigonometrischer Werte und gibt eine Realzahl aus. Um Gradzahlen in Radianangaben umzuwandeln, multiplizieren Sie die Gradzahl mit PI. (PI = 3.141593.) Dann teilen Sie das Ergebnis durch 180.

Anmerkung: Alle trigonometrischen Funktionen von ST BASIC erfordern eine Angabe der Winkel in Radian.

Beispiel:

```
Ok 10 RADIAN! = 34
Ok 20 TANGENTE! = TAN(RADIAN!)
Ok 30 PRINT TANGENTE!
Ok RUN
-.6235
Ok
```

TRACE 20,40
TRACE 20-40
TRACE -40

Syntax: TRACE [<Zeilenangaben-Liste>]

Effekt: Verfolgt den Programmlauf Zeile für Zeile und druckt selektiv die gesamte Zeile aus.

Erklärung:

Sie können den Befehl TRACE für die Aufdeckung und Behebung von Programmierfehlern verwenden, um die Programmzeilen während ihrer Ausführung darstellen zu lassen.

TRACE zeigt jede Programmzeile an, bevor sie ausgeführt wird.

TRACE 20, 40 stellt die Zeilen 20 und 40 jedesmal dar, wenn sie ausgeführt werden.

TRACE 20-40 druckt die Zeilen 20 bis 40 bei jeder Ausführung aus.

UNTRACE beendet den TRACE-Modus.

Lesen Sie hierzu auch unter TRON und FOLLOW nach.

Beispiel:

```
Ok 10 FOR X = 1 TO 2
Ok 20 N = N + 1
Ok 30 B = B + 1
Ok 40 PRINT N
Ok 50 PRINT B
Ok 60 NEXT X
Ok RUN



O  TRACE
O  RUN
T  ) FOR X = 1 TO 2
T 20 N = N + 1
T 30 B = B + 1
T 40 PRINT N
  1
T 50 PRINT B
  1
T 60 NEXT X
T 20 N = N + 1
T 30 B = B + 1
```

```
T 40 PRINT N
  1
T 50 PRINT B
  2
T 60 NEXT X
Ok UNTRACE
Ok
```

TROFF **TROFF** **BEFEHL**
TROFF 10, 40
TROFF 10-40
TROFF -40

Syntax: TROFF [<Zeilenangabe-Liste>]

Effekt: TROFF schaltet den Befehl TRON ab.

Erklärung:

TROFF schaltet den Befehl TRON entweder vollständig, oder lediglich für bestimmte Programmzeilen ab.

Lesen Sie hierzu auch unter TRON nach.

TRON **TRON** **BEFEHL**
TRON 20,40
TRON 20-40
TRON -40

Syntax: TRON [<Zeilenangabe-Liste>]

Effekt: Verfolgt selektiv den Programmlauf und druckt die Zeilennummern aus.

Erklärung:
TRON wird für die Fehleraufdeckung und -behebung verwendet, um den Programmlauf zeilenweise nachvollziehen zu können.

TRON stellt jede Programmzeile während ihrer Ausführung dar und behält die Übersicht über die Werte von Variablen. Die Zeilenangabe erfolgt in eckigen Klammern.

Mit TROFF wird der Befehl TRON abgeschaltet.

Lesen sie hierzu auch unter TRACE und FOLLOW nach.

Beispiel:

```
Ok 10 FOR X = 1 TO 3
Ok 20 N =  N + 1
Ok 30 B =  B + 1
Ok 40 PRINT N
Ok 50 PRINT B
Ok 60 NEXT X
Ok RUN
  1
  1
  2
  2
  3
  3
Ok TRON
Ok RUN
[10]
[20]
[30]
[40]  1   (erscheint im Ausgabefenster)
[50]  1   (erscheint im Ausgabefenster)
[60]
[20]
[30]
[40]  2   (erscheint im Ausgabefenster)
[50]  2   (erscheint im Ausgabefenster)
[60]
[20]
[30]
[40]  3   (erscheint im Ausgabefenster)
[50]  3   (erscheint im Ausgabefenster)
[60]
Ok TROFF
Ok
```

UNBREAK **UNBREAK** **BEFEHL**
UNBREAK 20, 50
UNBREAK -50
UNBREAK 20-50

Syntax: UNBREAK [<Zeilenangabe-Liste>]

Effekt: Widerruft selektiv einen BREAK-Befehl.

Erklärung:

UNBREAK widerruft den Befehl BREAK entweder vollständig oder für die angegebenen Zeilen.

Lesen Sie hierzu auch unter BREAK nach.

UNFOLLOW **UNFOLLOW** **BEFEHL**
UNFOLLOW X,Y

Syntax: UNFOLLOW [<Variable<],[<Variable>]

Effekt: Widerruft den Befehl FOLLOW entweder vollständig oder für die angegebenen Variablen.

Erklärung:

UNFOLLOW hebt den Befehl FOLLOW entweder vollständig, oder lediglich für die angegebenen Variablen auf.

Lesen Sie hierzu auch unter FOLLOW nach.

UNTRACE **UNTRACE** **BEFEHL**
UNTRACE 10, 40, 70
UNTRACE 10-40
UNTRACE -40

Syntax: UNTRACE [<Zeilenangabe-Liste>]

Effekt: Widerruft den Befehl TRACE.

Erklärung:

UNTRACE widerruft den Befehl TRACE entweder vollständig oder für angegebene Zeilennummern.

Lesen Sie hierzu auch unter TRACE nach.

VAL **X = VAL(A$)** **FUNKTION**

Syntax: X = VAL(<Ziffernstring-Ausdruck>)

Effekt: Durchsucht einen Zeichen-String und wandelt die Zeichen in Realzahlen um.

Erklärung:

VAL durchläuft eine Zeichenkette von links nach rechts und überspringt dabei vorangestellte Leerzeichen, Tabulatorenstops und Zeilenvorschübe, bis das Ende der Zeichenkette erreicht, bzw. bis ein Zeichen gefunden wurde, das keiner Ziffer entspricht. VAL durchsucht Strings in derselben Weise, in der die Anweisung INPUT# in numerischen Variablen liest.

Ist das erste Zeichen des Strings kein gültiger Teil einer Zahl, gibt VAL eine Null aus.

VAL ist das Gegenstück zu STR$.

Beispiel:

```
Ok 10 READ ID$
Ok 20 IF VAL(ID$) < 300 THEN 30
Ok 30 VERFALLDATUM$ = "1. JAN. 1986"
Ok 40 IF VAL(ID$) > 300 THEN 50
Ok 50 VERFALLDATUM$ = "1. JAN. 1990"
```

VARPTR **X = VARPTR(Y)** **FUNKTION**
X = VARPTR(#1)

Syntax: X = VARPTR(<Variable>)
X = VARPTR(#<Dateinummer>)

Effekt: Gibt die Adresse einer Variablen aus.

Erklärung:

Sie können VARPTR dazu verwenden, um die Adresse von Variablen zu erfahren, die an eine Maschinensprache-Unterroutine übergeben werden sollen. Die Variable kann beliebiger Art (also auch ein Array) sein. Sie müssen ihr jedoch einen Wert zugewiesen haben, bevor Sie über VARPTR ihre Adresse erfahren können. VARPTR gibt einen Wert aus, der der absoluten Adresse des ersten Bytes der bezeichneten Variablen entspricht.

Im Falle einer Datei entspricht die Dateinummer der Kennzahl, die Sie der Datei beim Öffnen zugewiesen hatten. VARPTR gibt die Startadresse des Eingabe-/Ausgabespeichers der Datei aus.

Beispiel:

```
Ok 50 X = VARPTR(MATERIAL)
```

VDISYS **VDISYS(1)** **FUNKTION**

Syntax: VDISYS(<Test-Argument<)

Effekt: Ermöglicht dem Anwender, Zugriff auf das VDI Interface des Betriebssystemes zu nehmen.

Erklärung:

Um Zugriff auf das VDI Interface zu nehmen, müssen Sie die Arrays CONTRL, INTIN UND PTSIN mit den entsprechenden Werten POKEn, bevor Sie VDISYS aufrufen. Über die Arrays INTOUT und PTSOUT kann eine Ausgabe vom VDI-Level aus erfolgen.

Beispiel:

```
Ok 10 REM KREIS BEI 50,50 MIT RADIUS 25
Ok 20 COLOR 1,1,1,1,1 :FULLW 2
Ok 30 POKE CONTRL,11
Ok 40 POKE CONTRL+2,3
Ok 50 POKE CONTRL+6,0
Ok 60 POKE CONTRL+10,4
Ok 70 POKE PTSIN,50
Ok 80 POKE PTSIN+2,50
Ok 90 POKE PTSIN+8,25
Ok 100 VDISYS(1)
```

Syntax: WAIT <Datenkanal-Nummer>,<Integer-Ausdruck> [,<Integer-Ausdruck>]

Effekt: Hält das Programm an und wartet darauf, daß vom I/O- Datenkanal ein Bit-Muster entwickelt wird.

Erklärung:

WAIT unterbricht einen Programmlauf solange, bis in einem Eingabekanal des Rechners ein vorgegebenes Bit-Muster erstellt wurde. Der logische Operator XOR überprüft die Daten des Kanals daraufhin, ob sie dem optional anzugebenden zweiten Integer- Ausdruck entsprechen. Wird für den zweiten Integer-Ausdruck keine Angabe gemacht, wird als Wert Null angenommen.

Der Operator AND überprüft die Daten dann gegen den ersten Ausdruck. Entspricht der Wert Null, wird die Programmausführung in einer Schleife wieder zurückgesetzt und wartet auf die nächsten Daten aus dem Kanal.

Findet WAIT kein Bit-Muster, das als Ergebnis Null ausweist, springt das Programm in eine Endlosschleife und Sie müssen den Rechner neu booten.

Beispiel:

```
Ok 100 WAIT 5,&H2,&H3
Ok 110 PRINT "ZAHL GEFUNDEN"
```

Syntax: WAVE <numerischer Ausdruck>, <numerischer Ausdruck>, <numerischer Ausdruck>, <numerischer Ausdruck>, <numerischer Ausdruck>,

Effekt: Mit WAVE können die Wellenformen in einer SOUND-Anweisung eingestellt werden.

Erklärung:

STIMME (voice) gilt für das Mixer-Register des Tongenerators. Eine Null auf den Bits 0-2 aktiviert die Stimmen 1-3. Eine Null auf den Bits 3-5 setzt "noise" für die Stimmen 1-3. Sie können mehr als eine Stimme gleichzeitig aktivieren.

HÜLLKURVE (envelope) ist das Hüllkurvengenerator-Register. Der Wert 1 auf den Bits 0-2 aktiviert die Hüllkurve für die Stimmen 1-3. Es können mehrere Hüllkurven gleichzeitig aktiviert werden.

FORM (shape) steht für das Register der Hüllkurvenform und Zykluseinstellung. Die Bits 0-3 werden gemäß der untenstehenden Tabelle gesetzt.

DAUER (period) legt die Zeitdauer der Hüllkurve fest.

VERZÖGERUNG (delay) dient zur Einstellung der Zeitintervalle (in 1/50 Sekunden-Erhöhungen), bevor ST BASIC mit dem Programmlauf fortfährt.

Das Registermodell des Tonerzeugungs-ICs

Register	Bit	B7	B6	B5	B4	B3	B2	B1	B0
R0	Kanal A Tonhöhe	untere 8 Bit (Ton A)							
R1		obere 4 Bit (A)							
R2	Kanal B Tonhöhe	untere 8 Bit (Ton B)							
R3		obere 4 Bit (B)							
R4	Kanal C Tonhöhe	untere 8 Bit (Ton C)							
R5		obere 4 Bit (C)							
R6	Rauschen	5 Bit Rauschperiode							
R7	Zuordnung	ein/aus I/OB	ein/aus I/OA	Rauschen C	Rauschen B	Rauschen A	Ton C	Ton B	Ton A
R8	Amplitude A				M	L3	L2	L1	L0
R9	Amplitude B				M	L3	L2	L1	L0
R10	Amplitude C				M	L3	L2	L1	L0
R11	Hüllkurven Frequenz	untere 8 Bit (Feinabstimmung)							
R12	Hüllkurven Frequenz	obere 8 Bit (Grobabstimmung)							
R13	Hüllkurve					E3	E2	E1	E0
R14	E/A-Port A	8 Bit Parallelport A							
R15	E/A-Port B	8 Bit Parallelport B							

Syntax: WEND

Effekt: Kennzeichnet das Ende einer WHILE/WEND-Schleife.

Erklärung:

WEND kann nur in Verbindung mit WHILE verwendet werden. Es übergibt die Programmausführung wieder zurück an die WHILE-Anweisung. Ein verschachteltes WEND nimmt Bezug auf die nächstgelegene WHILE-Anweisung.

Lesen Sie hierzu auch unter WHILE nach.

Beispiel:

```
Ok 10 X=8
Ok 20 WHILE X
Ok 30 PRINT "$";
Ok 40 X=X-1
Ok 50 WEND
Ok 60 END
Ok RUN
$$$$$$$$
Ok
```

Syntax: WHILE <logischer Ausdruck<

Effekt: Stellt eine Bedingung auf, die eine WHILE/WEND- Schleife steuert.

Erklärung:

WHILE beginnt eine WHILE/WEND-Schleife, die solange ausgeführt wird, bis der logische Ausdruck falsch ist (z.B. 0). Die Anweisungen zwischen WHILE und WEND werden ausgeführt, solange der konditionale Ausdruck innerhalb der WHILE-Anweisung wahr ist.

Die Anweisung WEND am Ende der Schleife gibt die Programmausführung wieder zurück an die WHILE-Bedingung. Die Bedingung der WHILE-Schleife wird zahlenmäßig berechnet und die Schleife wird solange erneut ausgeführt, bis die Bedingung nicht mehr wahr ist (0). Sobald die Bedingung falsch ist, wird der Programmlauf bei der Anweisung hinter WEND aufgenommen.

Sie können WHILE/WEND-Schleifen verschachteln. Jedes WEND gilt für das nächstgelegene WHILE. Die Verwendung von WHILE ohne dazugehöriges WEND oder umgekehrt verursacht eine Fehlermeldung.

Lesen Sie hierzu auch unter WEND nach.

Beispiel:

```
Ok 10 M=10
Ok 20 P=5
Ok 30 WHILE M>P
Ok 40 PRINT "ZAEHLSCHLEIFE"
Ok 50 M=M-1
Ok 60 WEND
Ok 70 END
Ok RUN
ZAEHLSCHLEIFE
ZAEHLSCHLEIFE
ZAEHLSCHLEIFE
ZAEHLSCHLEIFE
ZAEHLSCHLEIFE
Ok
```

Syntax: WIDTH [LPRINT] <Integer-Ausdruck>

Effekt: Stellt die Zeilenbreite für den Bildschirm oder Drucker ein.

Erklärung:

Die vorgegebene Zeilenbreite für den Bildschirm und den Drucker liegt bei 72 Zeichen. Diese Voreinstellung kann über WIDTH geändert werden.

Der Integer-Ausdruck entspricht der Zeichenanzahl pro Zeile und muß zwischen 14 und 255 liegen. Mit der Option LPRINT können Sie die Zeilenbreite für den Drucker angeben. Andernfalls wird lediglich die Zeilenbreite für die Bildschirmdarstellung verändert.

Bei einem Ausdruck setzt ST BASIC vor jedes Zeichen, das normalerweise hinter dem vorgegebenen Zeilenende gedruckt werden würde, eine Zeilenschaltung. Um unerwünschte Zeilenschaltungen bei einem Ausdruck zu verhindern, sollten Sie die Zeilenbreite auf 255 Zeichen setzen. ST BASIC geht dann davon aus, daß das angeschlossene Gerät eine unbegrenzte Zeilenbreite hat und fügt keine Zeilenschaltungen ein.

Lesen Sie hierzu auch unter POS und LPOS nach.

Beispiel:

```
Ok 10 WIDTH 33
Ok 20 FOR I=1 TO 50
Ok 30 PRINT "-";
Ok 40 NEXT
Ok RUN
---------------------------------
----------------
Ok
```

Syntax: WRITE[<Ausdruck>],<Ausdruck>

Effekt: Gibt Daten an das Terminal aus.

Erklärung:

Wie PRINT sendet auch WRITE Ausgaben an den Bildschirm. Allerdings setzt WRITE Kommata zwischen einzelne Begriffe und Anführungszeichen um Strings.

Jeder Begriff wird auf dem Terminal durch ein Kommazeichen vom nachfolgenden Begriff abgesetzt.

String-Werte werden in Anführungszeichen gesetzt. Nach dem letzten Begriff springt der Cursor an den Anfang der nächsten Zeile.

WRITE zeigt eine Leerzeile am Terminal an, wenn Sie keine Auflistung von Ausdrükken für eine Ausgabe vorgeben.

Lesen Sie hierzu auch unter PRINT und PRINT# nach.

Beispiel:

```
Ok 100 X$="HALLO"
Ok 110 Z=010583
Ok 120 WRITE Z
Ok 130 WRITE
Ok 140 WRITE X$
Ok RUN
  10583

"HALLO"
Ok
```

Syntax: WRITE#[<Ausdruck>],<Ausdruck>

Effekt: Gibt Daten an eine sequentielle Datei aus.

Erklärung:

WRITE# ist von der Funktion her ähnlich wie WRITE. Allerdings werden mit WRITE# die Daten an eine sequentielle Datei, und nicht an das Terminal gesandt. Die Dateinummer ist die Kennzahl, die Sie der Datei beim Öffnen zugewiesen hatten. Sie müssen die Datei im O-Modus geöffnet haben.

WRITE# ist der Anweisung PRINT# vorzuziehen, wenn Sie beabsichtigen, die Daten über eine Reihe von INPUT#-Anweisungen wieder zurückzulesen. Die Ausgabe von WRITE# erfolgt in der für ein akkurates Zurücklesen der Daten erforderlichen Form.

Die Richtlinien für die Form der Ausdrücke entsprechen den Vorgaben für PRINT#.

Lesen Sie hierzu auch unter PRINT und PRINT# nach.

Beispiel:

```
Ok 10 KWS=34.275
Ok 20 K$="DURCHSCHNITTLICHE KILOWATTSTUNDEN PRO WOCHE"
Ok 30 WRITE #2,K$,KWS
```

Damit wird wie folgt auf die Diskette geschrieben:

```
"DURCHSCHNITTLICHE KILOWATTSTUNDEN PRO WOCHE",34.275
```

Schließen Sie nun die Datei und öffnen Sie sie neu. Lesen Sie die Datei dann mit:

```
Ok 40 INPUT#2,K$,KWS
```

"DURCHSCHNITTLICHE KILOWATTSTUNDEN PRO WOCHE" für K$ und 34.275 für B$

ANHANG D

FEHLERMELDUNGEN

Kennziffer	Meldung
2	Fehlerhafte Eingabe
3	RETURN-Anweisung erfordert entsprechendes GOSUB
4	READ-Anweisung hat zuwenig DATA-Werte
5	Funktionsaufruf nicht erlaubt
6	Zahl zu groß
7	Zu wenig Speicherplatz
8	Anweisung oder Befehl bezieht sich auf nicht vorhandene Zeile
9	Unterbereich bezieht sich auf ein Element außerhalb des Arrays
10	Array mehr als einmal definiert
11	Division durch Null nicht möglich
12	Anweisung im Direkt-Modus ungültig
13	Werte stimmen vom Typ her nicht überein
14	Undefinierter Fehler
15	String länger als 255 Zeichen
16	Ausdruck zu lang oder zu komplex
17	CONT arbeitet nur im BREAK-Modus
18	Funktion muß erst über DEF FN definiert werden
19	Undefinierter Fehler
20	RESUME-Anweisung vor Aufrufen der Fehler-Routine
21	Nicht belegt
22	Ausdruck enthält Operator ohne nachfolgenden Operanden
23	Programmzeile zu lang
24 - 29	Nicht belegt
30	Ungültige Fenster-Nummer
31	Argument außerhalb des gültigen Bereiches
32	Befehl kann im Editor nicht ausgeführt werden
33	Zeile zu komplex
34 - 49	Nicht belegt
50	FIELD-Anweisung verursacht Überlauf
51	Ungültige Geräte-Nummer
52	Dateiname oder -nummer ungültig
53	Datei im angegebenen Laufwerk nicht gefunden
54	Ungültiger Datei-Modus
55	Löschen (KILL) oder Öffnen (OPEN) geöffneter Dateien nicht möglich
56	Undefinierter Fehler
57	Disketten-Eingabe/Ausgabefehler
58	Datei bereits vorhanden
59 - 60	Nicht belegt
61	Diskette voll

62	Dateiende erreicht
63	Datensatz-Nummer in PUT oder GET größer als 32767 oder 0
64	Ungültiger Dateiname
65	Ungültiges Zeichen in der Programmdatei
66	Programmdatei enthält Anweisung ohne Zeilennummer
67 - 98	Nicht belegt
99	--Break--
100	Undefinierter Fehler
101	Zu viele Programmzeilen
102	ON-Anweisung außerhalb des gültigen Bereiches
103	Ungültige Zeilennummer
104	Variablen-Eingabe erforderlich
105	Undefinierter Fehler
106	Zeilennummer existiert nicht
107	Zahl zu groß für eine Integer
108	Eingegebene Daten ungültig. Eingabe ab dem ersten Begriff neu beginnen
109	STOP
110	Unterroutinen-Aufruf zu verschachtelt
111	Ungültige BLOAD-Datei
112 - 201	Nicht belegt
202	Befehl hier nicht möglich
203	Zeilennummernangabe erforderlich
204	FOR-Anweisung ohne abschließendes NEXT oder WHILE-Anweisung ohne abschließendes WEND
205	NEXT-Anweisung ohne vorhergehendes FOR oder WEND-Anweisung ohne vorhergehendes WHILE
206	Komma fehlt
207	Klammer fehlt
208	OPTION BASE muß 0 oder 1 sein
209	Abschluß der Anweisung erforderlich
210	Zu viele Argumente in der Liste
211	Nicht belegt
212	Neudefinition von Variablen nicht möglich
213	Funktion mehr als einmal definiert
214	Sprung in eine Schleife nicht möglich
215 - 220	Nicht belegt
221	Systemfehler #X, Neustart erforderlich
222	Programm läuft nicht
223	Zu viele FOR-Schleifen

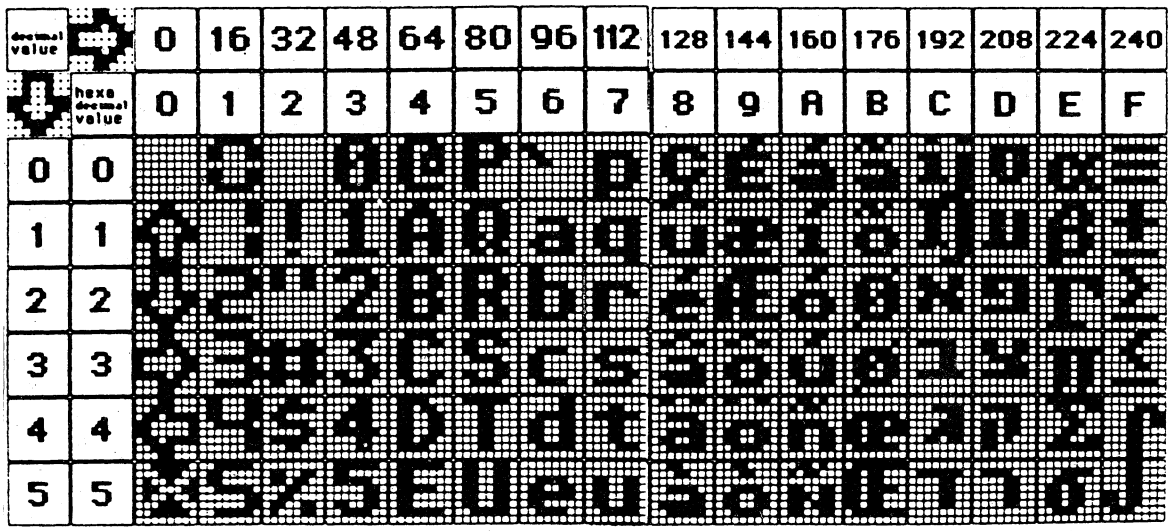
ANHANG E

DER ASCII-ZEICHENSATZ DES ST COMPUTERS

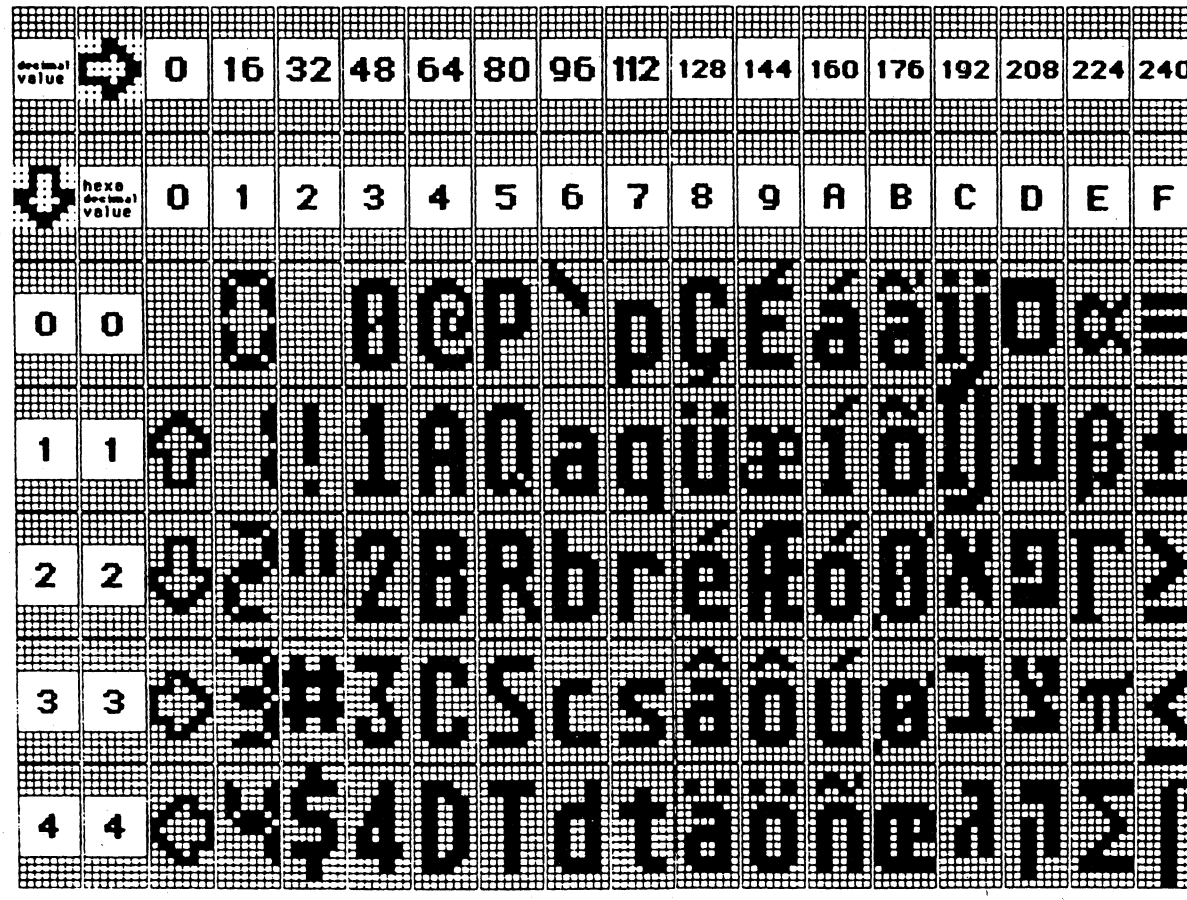
In den nachfolgenden Tabellen werden sämtliche Zeichensätze aufgeführt, die Ihnen mit dem ST Computer zur Verfügung stehen. Um eines dieser Zeichen in ST BASIC zu verwenden, geben Sie das nachstehende Programm ein und starten Sie es:

```
  5     'AUFLISTUNG ALLER ST ASCII-ZEICHEN
  6     'UND IHRER CODES
 10     FULLW 2:CLEARW 2
 20     GOTOXY 1,2:?"LISTE DER ST ASCII-ZEICHEN":
        GOTOXY 0,4
 30     P=0:1=0
 40     FOR C=1 TO 255
 50     IF P>4 THEN P=0:1=1+1:?
 60     IF 1=10 THEN GOTOXY 1,14: INPUT "WEITER MIT
        [RETURN] ... ",A$
 70     IF 1=10 THEN 1=0:GOTOXY 0,4
 80     IF C=10 THEN ? "10=[RETURN]";:GOTO 120
 90     IF C=7 THEN ? "7=[GLOCKE]";:GOTO 120
100     IF C=251 THEN GOTOXY 0,124
110     ? C;"= ";CHR$(C);" ";
120     P=P+1
130     NEXT C
140     GOTOXY 1,16: INPUT "BEENDEN MIT [RETURN] ... ",A$
150     END
```

Es gibt zwei Zeichen-Tabellen. Die erste gilt für 8x16 Zeichen, die zweite entsprechend für 16x16 Zeichen. Die verschiedenen Zeichensatz-Größen werden für die unterschiedlichen Bildschirmauflösungen verwendet.




6 6
7 7
8 8
9 9
10 A
11 B
12 C
13 D
14 E
15 F


decimal value
hexa decimal value
0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240
0 1 2 3 4 5 6 7 8 9 A B C D E F
0 0
1 1
2 2
3 3
4 4


5 5
6 6
7 7
8 8
9 9
10 A
11 B
12 C
13 D
14 E
15 F

ANHANG F

ASSEMBLER-SPRACHMODULE

Die CALL-Anweisung in ST BASIC ermöglicht eine Verwendung von Assembler-Sprachmodulen. Um ein Modul zu verwenden, müssen Sie dieses über eine BLOAD-Anweisung in den Arbeitsspeicher laden. Dann muß die Modul-Startadresse einer Variablen zugeordnet, und das Modul über BASIC mit CALL aufgerufen werden (wobei alle notwendigen Parameter übertragen werden).

Parameter werden in folgender Weise von BASIC an Assemblerprogramme übergeben: Das Maschinensprache-Modul sucht nach 2 Parametern auf dem Anwender-Stack (A7). Der erste Parameter ist eine 2-Byte Integerzahl, durch die die Anzahl der zu übergebenden Parameter vorgegeben wird. (Im untenstehenden Beispiel handelt es sich um drei Parameter.) Der zweite Parameter ist ein 4-Byte Zeiger auf ein Array, das die Parameter enthält. Jeder Parameter im Array belegt 8 Bytes, unabhängig davon, welcher Art er ist. Handelt es sich um eine String-Variable, ist der 8-Byte Wert ein Zeiger zu dem String.

Bevor es zum BASIC zurückkehrt, kann das Assemblerprogramm alle Parameter, die an BASIC übergeben werden sollen, in einer vorgegebenen Speicherstelle ablegen. Später kann das BASIC- Programm über PEEK auf diese Parameter zugreifen.

Beispiel:

```
500 DIM A$(8):I%=70:X=22
510 LISTE=18566: 'STARTADRESSE DES ASSEMBLER- SPRACHCODES
530 CALL LISTE(I%, A$, X)
```

ANHANG G

ABGELEITETE FUNKTIONEN

Abgeleitete Funktion	Terminologie der ST BASIC-Funktion
Sekante	DEF FNSEC(X)=1/COS(X)
Cosekante	DEF FNCSC(X)=1/SIN(X)
Inverser Sinus	DEF FNARCSIN(X)=ATN(X/SQR(-X*X+1))
Inverser Cosinus	DEF FNARCCOS(X)=-ATN(X/SQR(-X*X+1) +KONSTANTE)
Inverse Sekante	DEF FNARCSEC(X)=ATN(SQR(X*X-1))+ (SGN(X-1)*KONSTANTE)
Inverse Cosekante	DEF FNARCCSC(X)=ATN(1/SQR(X*X-1))+ (SGN(X-1)*KONSTANTE)
Inverse Cotangente	DEF FNARCCOT(X)=ATN(X)+KONSTANTE
Hyperbolischer Sinus	DEF FNSINH(X)=(EXP(X)-EXP(X))/2
Hyperbolischer Cosinus	DEF FNCOSH(X)=(EXP(X)+EXP(-X)/2
Hyperbolische Tangente	DEF FNTANH(X)=-EXP(-X)/(EXP(X)+EXP (-X))*2+1
Hyperbolische Sekante	DEF FNSECH(X)=2/(EXP(X)+EXP(-X))
Hyperbolische Cosekante	DEF FNCSCH(X)=2/(EXP(X)-EXP(-X))
Hyperbolische Cotangente	DEF FNCOTH(X)=EXP(-X)/(EXP(X)-EXP (-X))*2+1
Inverser hyperbolischer Sinus	DEF FNARCSINH(X)=LOG(X+SQR(X*X+1))
Inverser hyperbolischer Cosinus	DEF FNARCCOSH(X)+LOG(X+SQR(X*X-1))
Inverse hyperbolische Tangente	DEF FNARCTANH(X)=LOG((1+X)/(1-X))/2
Inverse hyperbolische Sekante	DEF FNARCSECH(X)=LOG((SQR(-X*X+1) +1)/X)
Inverse hyperbolische Cosekante	DEF FNARCCSCH(X)=LOG((SGN(X)*SQR (X*X+1)+1)/X)
Inverse hyperbolische Cotangente	DEF FNARCCOTH(X)=LOG((X+1)/(X-1))/2

Anmerkung: In dieser Tabelle entspricht die Variable X in Klammern dem Wert oder Ausdruck, der über die abgeleitete Funktion berechnet werden soll. Sie können einen beliebigen Variablennamen verwenden, solange dieser dem zu berechnenden Wert oder Ausdruck entspricht.

ANHANG H

BEISPIELPROGRAMME

KÄSTCHEN

Das nachfolgende Programm ist ein interessantes Beispiel für die Verwendung der Anweisung RND mit Farb-Grafiken. Das Programm wurde für Low-Resolution geschrieben.

```
 10  'SYMMETRISCHES AUSFÜLLEN VON KÄSTCHEN
 20  randomize 0:c=0
 30  color 1,0,1,1,1:fullw 2:clearw 2
 40  for x=18 to 284 step 19
 50  linef x,0,x,166
 60  next x
 70  for y=13 to 153 step 14
 80  linef 0,y,303,y
 90  next y
100  c=c+1:if c=16 then c=1
110  color 1,c,1
120  col=int(rnd*16)*19+9:row=int(rnd*12)*14+7
130  fill col,row,1
140  if col>151 then cenc=col-151:fill col-(cenc*2),row,1
150  if col<152 then colh=302-col:fill colh,row,1
160  if row>82 then rowh=row-((row-82)*2):fill col,rowh,1
170  if row<83 then rowh=164-row:fill col,rowh,1
180  if col>151 then fill col-(cenc*2),rowh,1 else fill colh,rowh,1
190  goto 100
```

GEMUSTERTE KREISE

Bei diesem Programm wird ein Kreis gezeichnet und in einzelne Segmente unterteilt. Die einzelnen Kreissegmente werden dann mit verschiedenen Mustern ausgefüllt. Um das Programm abzuwandeln, können Sie Zeile 120 folgendermaßen abändern:

```
120  pellipse x,y,x,y,b,b+100
```

```
 10  'KREIS MIT 36 GEMUSTERTEN SEGMENTEN
 20  color 1,0,1,1,1:fullw 2:clearw 2
 30  if peek(systab)=1 then 60
 40  if peek(systab)=2 then 70
 50  goto 80
 60  x=306:y=172:s=170:goto 90
 70  x=304:y=83:s=182:goto 90
 80  x=151:y=83:s=91
 90  a=24:i=2:b=0
100  for p=1 to a
110  color 1,1,1,p,i
120  pcircle x,y,s,b,b+100
```

```
130  b=b+100
140  next p
150  if i=1 then end
160  i=3:a=12:goto 100
```

GEMUSTERTES RASTER

In diesem Programm wird die Bildschirmauflösung automatisch ausgewählt. Dann werden 36 verschiedene Füllmuster dargestellt.

```
 10  'RASTER MIT 36 VERSCHIEDENEN FUELLMUSTERN
 20  color 1,0,1,1,1:fullw 2:clearw 2
 30  if peek(systab)=1 then 60
 40  if peek(systab)=2 then 70
 50  goto 80
 60  x=102:y=56:a=28:b=308:c=56:d=51:e=561:f=102:goto 90
 70  x=102:w=28:a=14:b=154:c=28:d=51:e=561:f=102:goto 90
 80  x=51:y=28:a=14:b=154:c=28:d=25:e=280:f=51
 90  for x=f to e-d step f
100  linef x,0,x,345
110  next x
120  for y=c to b-a step c
130  linef 0,y,615,y
140  next y
150  i=2:p=1
160  for y=a to b step c
170  for x=d to e step f
180  color 1,1,1,p,i:fill x,y,1
190  p=p+1:if p=25 then p=1:i=i+1
200  if i=4 then end
210  next x,y
```

DEMO FÜR NIEDRIGE AUFLÖSUNG

Eine interessante Demonstration über Formen und Farben in niedriger Auflösung.

```
 10  color 1,0,1,1,1:fullw 2:clearw 2
 20  KREIS: c=1
 30  for b=0 to 3360 step 240
 40  color 1,c,1
 50  pcircle 151,83,91,b,b+240
 60  c=c+1
 70  next b
 80  gosub VERZOEGERUNG
 90  OVAL: c=1
100  for b=0 to 3360 step 240
110  color 1,c,1
120  pellipse 151,83,151,83,b,b+240
130  c=c+1
140  next b
150  gosub VERZOEGERUNG
160  FILLPTNS: c=1:a=24:i=2
170  for p=1 to a
180  clearw 2
190  for x=61 to 244 step 61
200  linef x,0,x,166
210  next x
220  for y=55 to 110 step 55
230  linef 0,y,303,y
240  next y
250  y=2
260  for x=30 to 270 step 60
270  color 1,c,1,p,i
280  fill x,y,1
290  c=c+1:if c=16 then c=1
300  next x
310  y=y+55:if y=167 then 330
320  goto 260
330  next p
340  if i=3 then 360
350  a=12:i=3:goto 170
360  gosub VERZOEGERUNG
370  FARBIGERKREIS: c=1:r=91
380  for b=0 to 3600 step 200
390  color 1,c,1
400  pcircle 151,83,r,b,b+200
410  c=c+1:if c=16 then c=1
420  next b
430  r=r-1:if r=0 then 450
440  goto 380
450  gosub VERZOEGERUNG
```

```
460  FARBIGEELLIPSE:c=1:x=151:y=83
470  for b=0 to 3600 step 240
480  color 1,c,1
490  pellipse 151,83,x,y,b,b+240
500  c=c+1:if c=16 then c=1
510  next b
520  x=x-2:y=y-2:if y=3 then 540
530  goto 470
540  gosub VERZOEGERUNG
550  end
560  VERZOEGERUNG: for z=1 to 3000:next
570  color 1,0,1,1,1:clearw 2
580  return
```

DEMO FÜR HOHE AUFLÖSUNG

Dieses Programm zeigt die Möglichkeiten Ihres hochauflösenden Monochrom-Monitors.

```
 10  fullw 2:clearw 2
 20  QUADRATE: a=2:b=3:1=61:w=56
 30  x=a:y=b
 40  linef x,y,x+1,y
 50  linef x+1,y,x+1,y+w
 60  linef x+1,y+w,x,y+w
 70  linef x,y+w,x,y
 80  x=x+61
 90  if x>600 then x=a:y=y+56
100  if y>320 then 120
110  goto 40
120  a=a+2:b=b+2:1=1-4:w=w-4
130  if w<0 then 150
140  goto 30
150  gosub VERZOEGERUNG
160  LINIEN: x=0:y=0
170  while x<614
180  linef 307,172,x,y
190  x=x+5
200  wend
210  while y<344
220  linef 307,172,x,y
230  y=y+3
240  wend
250  while x>0
260  linef 307,172,x,y
270  x=x-5
```

```
280  wend
290  while y>0
300  linef 307,172,x,y
310  y=y-3
320  wend
330  gosub VERZOEGERUNG
340  ENTWURF: x1=1:x2=614:y1=1:y2=343
350  linef x1,y1,x2,y1
360  linef x2,y1,x2,y2
370  linef x2,y2,x1,y2
380  linef x1,y2,x1,y1
390  x1=x1+2:x2=x2-2:y1=y1+2:y2=y2-2
400  if y2>-22 then 350
410  gosub VERZOEGERUNG
420  end
430  VERZOEGERUNG: for z=1 to 5000:next
440  clearw 2:return
```

TRIGONOMETRISCHE GRAFIKEN

Mit diesem Programm können Sie beliebige trigonometrische Funktionen grafisch veranschaulichen.

```
 10  'TRIG GRAPHS
 20  'VON ROB COLLIER
 30  pi=3.1415926
 40  fullw 2:color 1,0,1:clearw 2
 50  BILDSCHIRM:
 60  if peek(systab)=4 then goto LOW
 70  if peek(systab)=2 then goto MEDIUM
 80  if peek(systab)=1 then goto HIGH
 90  INIT: t=0:1=0
100  lng=r/4:inc=pi/lng:off=b/4
110  FUNKTION: value=-2*pi
120  clearw 2
130  print"Funktion auswählen:":print
140  print "1) Sinus"
150  print "2) Cosinus"
160  print "3) Tangente"
170  print "4) Cosekante"
180  print "5) Sekante"
190  print "6) Cotangente"
200  print:input wahl
210  if wahl>0 and wahl<7 then goto GRAFIK
220  ?"wählen Sie eine dieser Zahlen aus."
230  goto FUNKTION
240  ZEICHNEN:
250  value=-2*pi
260  x=1:x1=1:y1=b/2
```

```
260  x=1:x1=1:y1=b/2
270  on wahl gosub
SINUS,COSINUS,TANGENTE,COSEKANTE,SEKANTE,COTANGENTE
280  y=off*y:y=b/2-y
290  if y<t or y>b goto SPRUNG
300  if x<1 or x>r goto SPRUNG
310  linef x1,y1,x,y
320  SPRUNG: x1=x
330  y1=y:x=x+1
340  value=value+inc
350  if value>2*pi then goto ENDE
360  goto 270
370  ENDE: input wait$
380  goto 120
390  GRAFIK: color 1,bg,gr:clearw 2
400  linef 1,b/2,r,b/2
410  linef r/2,t,r/2,b
420  color 1,bg,ln
430  goto ZEICHNEN
440  SINUS: y=sin(value):return
450  COSINUS: y=cos(value):return
460  TANGENTE: y=tan(value):return
470  COSEKANTE: hold=sin(value)
480  if hold=0 then return
490  y=1/hold:return
500  SEKANTE: hold=cos(value)
510  if hold=0 then return
520  y=1/hold:return
530  COTANGENTE:hold=tan(value)
540  if hold= 0 then return
550  y=1/ho d:return
560  LOW: ı =303:b=167
570  gr=2:ln =14:bg=4
580  goto IN T
590  MEDIU M: r=608:b=167
600  gr=1:ln =2:bg=3
610  goto INıT
620  HIGH: r= 615:b=343
630  gr=1:ln=  :bg=0
640  goto INIT
```

EFFEKTIVZINS-BERECHNUNG

Das nachstehende Programm kann für die Analyse Ihrer Finanzen verwendet werden.

```
 10  'Effektivzins-Berechnungsprogramm von Richard Lauck
 20  'Das Programm arbeitet mit einer Formel der Newton'schen Methode zum
     Schätzen von Wurzeln
 30  'Es verwendet Integralberechnungen mit Ypsilon, "Y", definiert in Zeile 100
 40  'Bei den Formeln wird vorausgesetzt, daß alle Zahlungen am Ende eines Zeit-
     raumes getätigt werden.
 50  clearw 2:fullw 2:?
 60  ? "LETZTE PAUSCHALZAHLUNG = ";:INPUT R
 70  ? "MONATLICHE RATE = ";:INPUT A
 80  ? "KAUFPREIS HEUTE = ";:INPUT C
 90  ? "ANZAHL DER RATEN = ";:INPUT N
100  Z=12:I=0.01:Y=0.01:K=0
110  ?:PRINT " EFFEKTIVER ZINSSATZ BEI: "
120  PRINT " "
130  PRINT "EINE LETZTE PAUSCHALZAHLUNG VON DM";R
140  PRINT "EINE MONATLICHE RATE VON DM";A
150  PRINT "BEI EINER RATENZAHL VON - ";N
160  GOSUB 250
170  F=F+5.0Y-03:F=100*F:F=INT(F):F=F/100
180  F1=F1+5.0Y-03:F1=100*F1:F1=INT(F1):F1=F1/100
190  I=I1:K=K+1
200  IF ABS(F)-Y>0 THEN 160
210  PRINT " "
220  X=Z*I:PRINT "DER EFFEKTIVE ZINSSATZ IST ";100*X;"%"
230  PRINT " "
240  END
250  T=(1+I)^N
260  F=C-R/T-A*(1-1/T)/I
270  T2=T*(1+I)
280  F1=R*N/T2+A*(1-1/T-I*N/T2)/I/I
290  I1=I-F/F1
300  RETURN
```

ZAHLENSPIEL

Dieses Programm erstellt ein selbständig antwortendes Zahlenspiel. Sie geben eine Zahl ein, der Computer wählt eine zwischen Ihrer Vorgabe und Null liegende Zahl aus, und Sie müssen diese Zahl erraten.

```
 10  'Ein Spiel, das Sie sich selbst leicht oder schwer machen können - von Rich Lauck.
 20  fullw 2:clearw 2
 30  goto 0,0
 40  ? " Spielen wir ZAHLEN RATEN?"
 50  ? " Geben sie eine Zahl ein und drücken Sie"
 60  ? " RETURN. Ich waehle eine Zahl zwischen"
 70  ?" Ihrer Zahl und ";
 80  ?"Null."
 90  ? " Also, geben Sie eine Zahl ein "
100  INPUT " und drücken Sie RETURN. ",TOP
110  ?:? "Versuchen Sie, meine Zahl zu erraten "
120  RANDOMIZE 0
130  ANTWORT =INT(RND*(TOP))
140  ?:? " Raten Sie und ich gebe Hinweise.":goto 180
150  ?:input " J für neues Spiel, andere Taste für ENDE. ",neu$:?
160  if neu$="j" or neu$="J" then 90
170  end
180  input frage
190  if frage > antwort then ?"Zahl zu klein, neuer Versuch.":goto 180
200  if frage > antwort then ?"Zahl zu groß, neuer Versuch.":goto 180
210  ? "Richtig geraten.":goto 150
```

KÄSTCHEN-DEMO

Dieses Farbprogramm in niedriger Auflösung verwendet AES und VDI, um mehrfarbige Kästchen an einer von Ihnen gewählten Bildschirmposition zu zeichnen.

AES (Application Environment Services) ist der Teil von GEM, der für die Drop-Down Menüs, Fenster und Dialogfelder zuständig ist. VDI (Virtual Device Interface) enthält die Grafik- und Textroutinen von GEM.

Befolgen Sie nachstehende Schritte, um das Programm einzusetzen:

1. Starten Sie das Programm mit RUN

2. Zeigen Sie mit Hilfe der Maus auf die Bildschirmposition, an der das Kästchen gezeichnet werden soll.

3. Drücken Sie die rechte Maustaste, um das Kästchen zu zeichnen.

4. Drücken Sie die linke Maustaste, um das Programm zu beenden.

```
 5  a# = gb
10  control = peek (a#)
20  global = peek(a# + 4) 30 gintin = peek(a# + 8)
```

```
  40  gintout = peek(a# + 12)
  50  addrin = peek(a# + 16)
  60  addrout = peek(a# + 20)
 100  clearw 2:fullw 2
1070  poke systab+24,1
1071  poke contrl,122:poke contrl+2,0:poke contrl+6,1
1072  poke intin,0:vdisys(1)
1074  mouse =1
1075  gemsys(79)
2000  x = peek(gintout + 2)
2010  y = peek (gintout + 4)
2020  key = peek (gintout + 6)
2025  if key = 2 then gosub 3000
2027  if key = 1 then poke systab+24,0:end
2028  if key=0 then gosub 3115
2030  goto 1075
3000  rem ***************************************
3010  rem Kästchen zeichnen mit vdi
3020  rem ***************************************
3022  color 1,(rnd*15)+1,1,rnd*25,2
3024  if mouse=0 then 3040
3030  mouse=0
3035  poke contrl,123:poke contrl+2,0:poke contrl+6,0
3037  vdisys(1)
3040  poke contrl,11
3050  poke contrl +2,2
3060  poke contrl +6,0
3070  poke contrl +10,1
3080  poke ptsin,x
3090  poke ptsin +2,y
3095  poke ptsin +4,x+50
3100  poke ptsin +6,y+50
3110  vdisys(1)
3112  return
3115  if mouse=1 then return
3116  poke contrl,122:poke contrl+2,0:poke contrl+6,0
3117  poke intin,0:vdisys(1)
3120  mouse =1: return
3130  end
```

CO26166 / G
Printed in Taiwan
K. I. 3. 1986