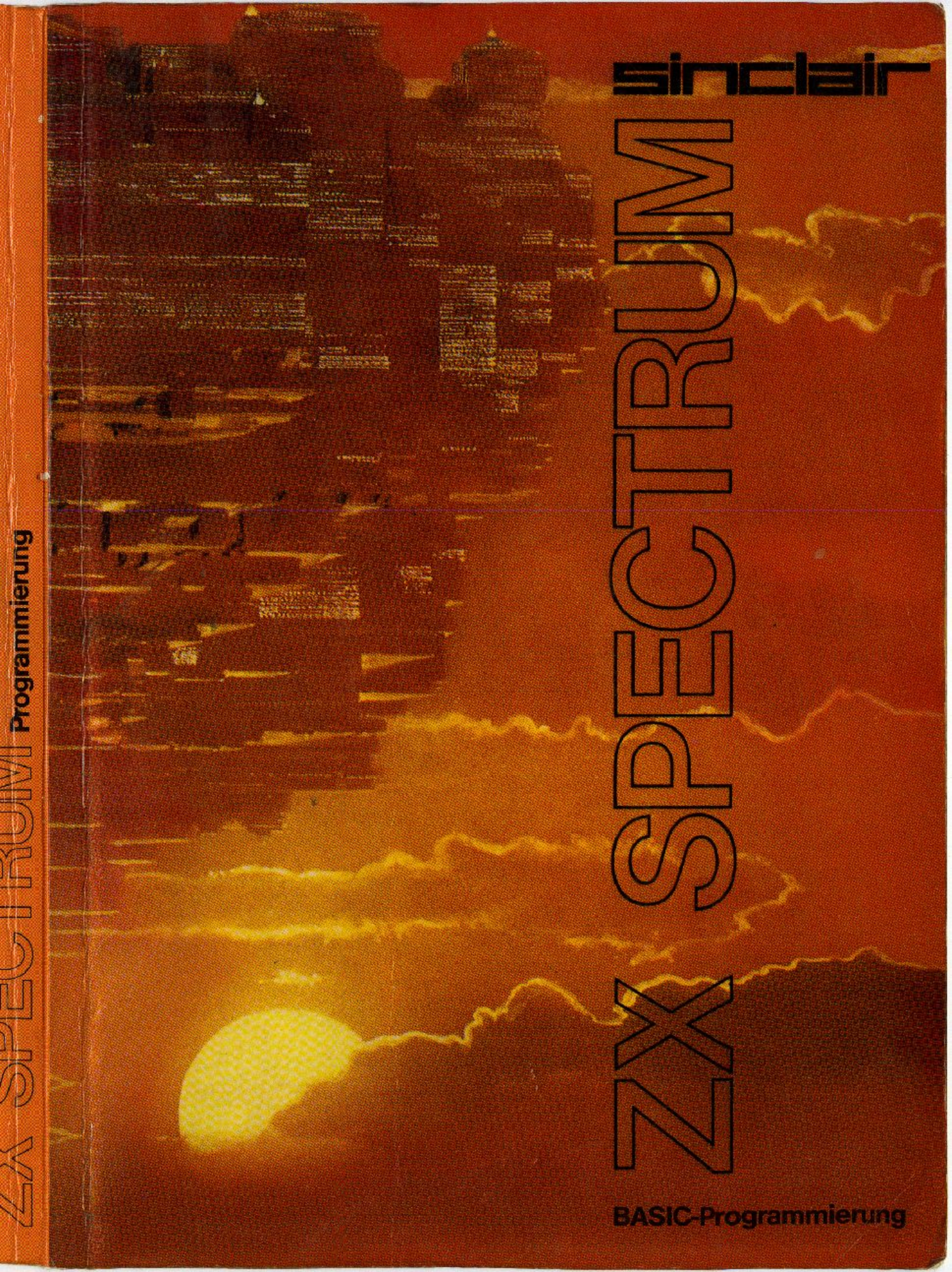

sinclair
ZX SPECTRUM
BASIC-Programmierung
ZX SPECTRUM Programmierung

Titel der englischen Originalausgabe
SINCLAIR ZX SPECTRUM BASIC PROGRAMMING

2. Auflage 1983

Übersetzung: T. Westermayr
Umschlaggrafik: John Harris of Young Artists
Herstellung: Werbeagentur Cooperation, München
ISBN 3-88945-011-3
Printed in Germany

sinclair

ZX SPECTRUM

Von Steven Vickers

BASIC-Programmierung

Inhalt:

KAPITEL 1

Einleitung

Ob Sie zuerst das Einführungsheft gelesen haben oder gleich nach diesem Band greifen, über folgende Punkte sollten Sie sich im klaren sein: Befehle werden sofort befolgt, Anweisungen beginnen mit einer Zeilennummer und werden für später gespeichert. Sie sollten außerdem die Befehle **PRINT**, **LET** und **INPUT** (bei allen Computern, die BASIC verwenden) und **BORDER**, **PAPER** und **BEEP** (beim Spectrum benutzt) kennen.

Dieses Handbuch wiederholt zu Beginn einige Dinge, die schon in der kleinen Anleitung stehen, aber mit mehr Einzelheiten. Sie erfahren genau, was Sie tun und was Sie nicht tun können. Am Ende des Kapitels finden Sie ein paar Übungen. Gehen Sie nicht darüber hinweg; viele davon veranschaulichen Dinge, die im Text stehen. Lesen Sie sie durch, machen Sie das, was Sie interessiert oder Punkte zu betreffen scheint, die Sie nicht ganz verstanden haben.

Wie auch immer – benutzen Sie auf alle Fälle häufig Ihren Computer. Wenn Sie die Frage drückt: "Was tut er, wenn ich das und das mache?", ist die Antwort leicht: Geben Sie das ein und sehen Sie selbst. Fragen Sie sich jedesmal, wenn das Handbuch Sie auffordert, etwas einzutippen: "Was könnte ich stattdessen schreiben?", und probieren Sie aus, was Ihnen einfällt. Je mehr Programme Sie selbst schreiben, desto besser verstehen Sie den Computer.

Am Ende des Bandes finden sich einige Anhänge. Dazu gehören Abschnitte über die Art, wie der Speicher aufgebaut ist, wie der Computer mit Zahlen umgeht, und eine Reihe von Programmbeispielen, um deutlich zu machen, was man mit dem ZX Spectrum alles anstellen kann.

Die Tastatur

ZX Spectrum-Zeichen umfassen nicht nur die Einzel*symbole* (Buchstaben, Ziffern, etc.), sondern auch die zusammengesetzten *Token* (Schlüsselwörter, Funktionsnamen, etc.). Diese werden allesamt nicht buchstabenweise, sondern direkt über die Tastatur eingegeben. Damit alle diese Funktionen und Befehle erreichbar sind, haben einige Tasten fünf oder mehr deutlich unterschiedene Bedeutungen. Zum einen Teil werden sie durch Umschalten (gleichzeitig mit der gewünschten Taste wird die Taste **CAPS SHIFT** oder **SYMBOL SHIFT** gedrückt), zum anderen dadurch erreicht, daß der Computer sich in einem jeweils anderen *Modus* (Betriebsart) befindet.

Der Modus wird angezeigt durch den *Cursor*, einen blinkenden Buchstaben. Er verweist darauf, wo das nächste Zeichen von der Tastatur eingeschoben wird.

Der K (für keyword = Schlüsselwort)-Modus tritt automatisch an die Stelle des L-Modus, wenn der Computer einen Befehl oder eine Programmzeile (statt **INPUT**-Daten) erwartet und von der Position des Cursors in der Zeile her weiß, daß er eine Zeilennummer oder ein Schlüsselwort zu erwarten hat. Das ist am Beginn der Zeile oder gleich nach **THEN** oder gleich nach **:** (außer in einem String). Bleibt eine Umschaltung aus, dann wird die nächste Taste entweder als Schlüsselwort (Schlüsselwörter stehen auf den Tasten) oder als Ziffer aufgefaßt.

Der L (für letter = Buchstabe)-Modus ist sonst die normale Betriebsart. Ohne Umschaltung wird die nächste Taste so gelesen, wie das Hauptsymbol auf ihr, bei Buchstaben in Kleinschrift.

Im K-, wie im L-Modus werden **SYMBOL SHIFT** und eine Taste aufgefaßt als das zusätzliche rote Zeichen auf der Taste, **CAPS SHIFT** mit einer Zifferntaste als die weiße Steuerfunktion über der Taste gelesen. **CAPS SHIFT** mit anderen Tasten beeinflußt die Schlüsselwörter im K-Modus nicht, im L-Modus verwandelt es die kleinen Buchstaben in große.

Der C (für capitals = Großbuchstaben)-Modus ist eine Abart des L-Modus; alle Buchstaben werden hier großgeschrieben. **CAPS LOCK** sorgt für die Umschaltung vom L-Modus in den C-Modus und wieder zurück.

Der E (für extended = erweitert)-Modus wird verwendet, um zusätzliche Zeichen zu erhalten, meistens Token. Er tritt dann ein, wenn beide Shift (Umschalt)-Tasten gleichzeitig gedrückt werden, und hält nur für einen Tastendruck an. In diesem Modus liefert eine Taste ein Zeichen oder Token (in grüner Schrift über der Taste) und ein zweites (unter der Taste in roter Schrift), wenn sie zusammen mit einer der beiden Umschalttasten gedrückt wird. Eine Zifferntaste liefert ein Token, wenn sie zusammen mit **SYMBOL SHIFT** gedrückt wird, im anderen Fall eine Farbsteuerungsfolge.

Der G (für graphics = Grafik)-Modus tritt ein, nachdem **GRAPHICS** (**CAPS SHIFT** mit **9**) gedrückt worden ist, und hält an, bis diese beiden erneut oder die **9** erneut gedrückt werden. Eine Zifferntaste liefert ein Grafikmosaikzeichen, ausgenommen **GRAPHICS** und **DELETE**, jede Buchstabentaste außer V, W, X, Y und Z liefert ein vom Benutzer wählbares Grafikzeichen.

Jede Taste, die länger als 2 bis 3 Sekunden niedergedrückt wird, läuft automatisch.

Tastatureingaben erscheinen wie geschrieben auf der unteren Bildschirmhälfte; jedes Zeichen (Einzelsymbol oder zusammengesetztes Token) wird unmittelbar vor dem Cursor eingesetzt. Der Cursor kann mit **CAPS SHIFT** und **5** nach links, mit **CAPS SHIFT** und **8** nach rechts bewegt werden. Das Zeichen vor dem Cursor kann mit **DELETE** (**CAPS SHIFT** und **0**) gelöscht werden. (Beachten Sie: Eine ganze Programmzeile kann auf einmal gelöscht werden durch Eingabe der Zeilennummer direkt gefolgt von **ENTER**).

Nach Druck auf **ENTER** wird die Zeile je nach Bedarf ausgeführt, ins Programm aufgenommen oder als **INPUT**-Daten behandelt, es sei denn, sie enthält einen Syntaxfehler. In diesem Fall erscheint neben dem Fehler ein blinkendes ?.

Mit der Eingabe von Programmzeilen wird in der oberen Bildschirmhälfte ein Listing angezeigt. Die letzte eingegebene Zeile heißt *laufende* Zeile und wird angezeigt durch das Symbol >; dieses kann bewegt werden durch die Tasten ▼ (**CAPS SHIFT** und **6**) und ▲ (**CAPS SHIFT** und **7**). Durch Drücken von **EDIT** (**CAPS SHIFT** und **1**) wird die laufende Zeile auf dem Bildschirm nach unten geholt und kann dort redigiert werden.

Wird ein Befehl ausgeführt oder ein Programm gefahren, erscheint die Ausgabe in der oberen Bildhälfte und bleibt dort, bis eine Programmzeile eingegeben oder **ENTER** mit einer leeren Zeile oder ▲ oder ▼ gedrückt wird. In der unteren Hälfte erscheint eine Meldung mit einem Code (Ziffer oder Buchstabe); die Bedeutung der Codes ergibt sich aus Anhang B. Die Meldung bleibt auf dem Bildschirm, bis eine Taste gedrückt wird (und den K-Modus anzeigt).

Unter bestimmten Umständen wird **CAPS SHIFT** mit der **SPACE**-Taste zusammen als **BREAK** benutzt. Das hält den Computer mit Meldung **D** oder **L** an. Das geschieht

- am Ende einer Anweisung, während ein Programm läuft, oder
- während der Computer den Kassettenrecorder oder den Drucker benutzt.

Der Fernseh-Bildschirm

Er besteht aus 24 Zeilen zu je 32 Zeichen und ist in zwei Hälften aufgeteilt. Die obere Hälfte verfügt über höchstens 22 Zeilen und zeigt entweder ein Listing oder eine Programmausgabe. Sobald die Anzeige in der oberen Hälfte die unterste Zeile erreicht hat, rollt sie um eine Zeile hoch. Der Computer zeigt die Meldung **scroll?** (Abrollen?) an. Druck auf die Tasten **N**, **SPACE** oder **STOP** hält das Programm mit der Meldung **D BREAK - CONT repeats** (D Unterbrechung - Forts. mit **CONT**) an; jede andere Taste setzt das Abrollen fort. Der untere Bildschirmteil wird verwendet für die Eingabe von Befehlen, Programmzeilen und **INPUT**-Daten sowie für die Anzeige von Meldungen. Der untere Teil hat anfangs zwei Zeilen (die obere leer), erweitert sich aber, um alles aufzunehmen, was eingegeben wird. Wenn die laufende Anzeigeposition in der oberen Bildschirmhälfte erreicht wird, führt weitere Ausdehnung dazu, daß die obere Hälfte aufwärts wegrollt.

KAPITEL 2

Grundbegriffe des Programmierens

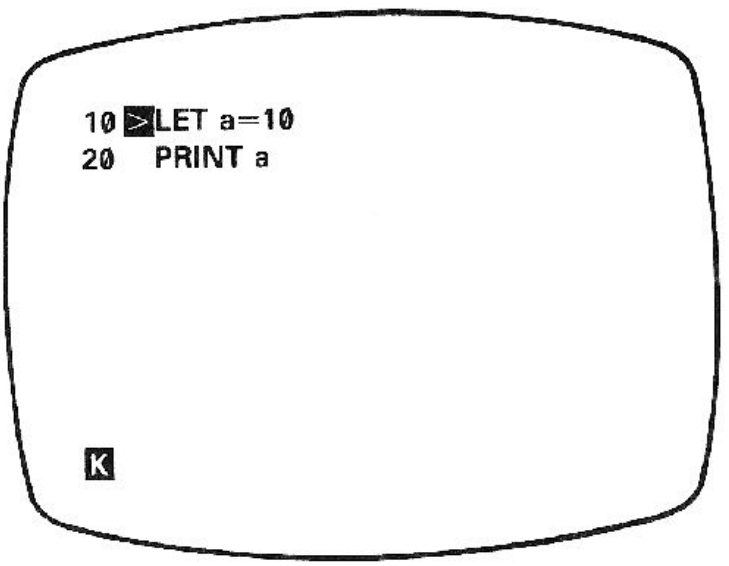
Zusammenfassung

Programme
Zeilennummern
Programme korrigieren mit ↑, ↓ und **EDIT**
RUN, LIST
GO TO, CONTINUE, INPUT, NEW, REM, PRINT
STOP in **INPUT**-Daten
BREAK

Tippen Sie diese zwei Zeilen eines Computerprogramms ein, um die Summe zweier Zahlen anzeigen zu lassen:

```
20 PRINT a
10 LET a=10
```

so daß der Bildschirm so aussieht:



Wie Sie schon wissen, wurden diese Zeilen nicht sofort ausgeführt, sondern als Programmzeilen gespeichert, weil sie mit Zeilennummern beginnen. Außerdem wird Ihnen hier aufgefallen sein, daß die Zeilennummern die Reihenfolge der Zeilen innerhalb des Programms bestimmen. Das ist besonders wichtig, wenn das Programm gefahren wird, zeigt sich aber auch an der Reihenfolge der Zeilen in dem Listing, das Sie jetzt auf dem Bildschirm sehen können.

Bis jetzt haben Sie erst eine Zahl eingegeben, also schreiben Sie jetzt

```
15 LET b=15
```

und geben das ein. Es wäre unmöglich gewesen, diese Zeile zwischen den ersten beiden einzuschieben, wenn sie statt 10 und 20 die Nummern 1 und 2 gehabt hätten (Zeilennummern müssen ganze Zahlen zwischen 1 und 9999 sein). Das ist der Grund, warum es bei der Ersteingabe eines Programms nützlich ist, zwischen den Zeilennummern Lücken zu lassen.

Jetzt müssen Sie Zeile 20 abändern zu

20 PRINT a+b

Sie könnten den Ersatzbefehl auch ganz schreiben, aber es ist einfacher die **EDIT**-Einrichtung zu nutzen, die im Einführungsheft beschrieben wird. Der ▶ in Zeile 15 wird Programmcursor genannt, und die Zeile, auf die er zeigt, ist die laufende Zeile. Das ist in der Regel die letzte Zeile, die Sie eingegeben haben, Sie können den Programmcursor aber mit den Tasten ▲ und ▼ aufwärts und abwärts bewegen. (Probieren Sie das aus, und lassen Sie den Programmcursor schließlich in Zeile 20 stehen.)

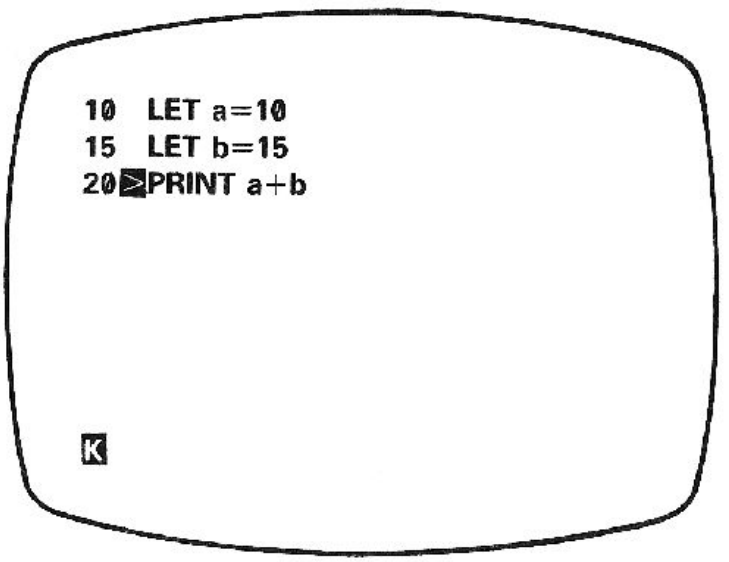
Wenn Sie die **EDIT**-Taste drücken, wird eine Kopie der laufenden Zeile unten am Bildschirm angezeigt – in Ihrem Fall eine Kopie von Zeile 20. Halten Sie die Taste ▶ gedrückt, bis der **L**-Cursor zum Zeilenende gelaufen ist, und tippen Sie dann

+b (ohne **ENTER**)

Die Zeile unten am Bildschirm sollte jetzt lauten

20 PRINT a+b

Drücken Sie **ENTER**. Die alte Zeile 20 wird durch die neue ersetzt, auf dem Bildschirm sieht es nun so aus:



Fahren Sie dieses Programm mit **RUN** und **ENTER**, dann wird die Summe angezeigt.
Lassen Sie das Programm ein zweitesmal laufen und geben Sie dann ein

PRINT a, b

Die Variablen sind noch da, obwohl das Programm fertig ist.
Eine nützliche Methode, um mit **EDIT** den unteren Bildschirmteil zu entfernen: Sie geben irgendein Kauderwelsch ein und kommen zu dem Schluß, daß Sie das doch nicht haben wollen. Ein Weg, das zu löschen, ist der, die **DELETE**-Taste gedrückt zu halten, bist die ganze Zeile gelöscht ist, aber es gibt noch einen anderen. Wenn Sie **EDIT** drücken, wird das Kauderwelsch unten am Bildschirm durch eine Kopie der laufenden Zeile ersetzt. Drücken Sie nun **ENTER**, wird die laufende Zeile unverändert ins Programm zurückgestellt, der untere Bildschirmteil ist leer.
Wenn Sie versehentlich eine Zeile eingeben, etwa

12 LET b=8

wird sie oben ins Programm gestellt, wo Sie Ihren Fehler erkennen. Um diese überflüssige Zeile zu löschen, tippen Sie nur

12 (natürlich mit **ENTER**)

Sie werden erstaunt feststellen, daß der Programmcursor verschwunden ist. Sie sollten sich vorstellen, daß er zwischen den Zeilen 10 und 15 versteckt ist; wenn Sie also ▲ drücken, springt er zu Zeile 10 hinauf, bei ▼ dagegen zu Zeile 15 hinunter.
Tippen Sie

12 (und **ENTER**)

Wieder ist der Programmcursor zwischen den Zeilen 10 und 15 versteckt. Wenn Sie jetzt **EDIT** drücken, kommt die Zeile 15 herunter: Sobald der Programmcursor zwischen zwei Zeilen steckt, holt **EDIT** die nächste Zeile nach der neuen Zeilennummer herunter. Geben Sie **ENTER** ein, um den unteren Bildschirmteil zu leeren.
Schreiben Sie nun

30 (und **ENTER**)

Dieses Mal ist der Programmcursor nach dem Programmende versteckt, und wenn Sie **EDIT** drücken, wird Zeile 20 heruntergeholt.
Als letztes schreiben Sie

List 15

Auf dem Bildschirm sehen Sie nun

15 ▶ LET b=5
20 PRINT a+b

Zeile 1Ø ist vom Bildschirm verwunden, befindet sich aber noch in Ihrem Programm – das läßt sich dadurch beweisen, daß Sie **ENTER** drücken. Die einzigen Wirkungen von **LIST 15** sind die, ein Listing zu liefern, das mit Zeile 15 beginnt, und den Programmcursor in Zeile 15 zu stellen. Wenn Sie ein sehr langes Programm haben, wird **LIST** vermutlich eine praktischere Methode sein, den Programmcursor zu bewegen, als ↓ und ↑.

Das veranschaulicht einen weiteren Nutzen von Zeilennummern: Sie dienen als Namen für die Programmzeilen, so daß Sie sich auf sie beziehen können, ganz ähnlich, wie Variable Namen haben.

LIST allein läßt das Listing am Programmbeginn anfangen.

Ein weiterer Befehl, den Sie im Einführungsheft kennengelernt haben, ist:

NEW

Das löscht alle alten Programme und Variablen im Computer. Geben Sie jetzt sorgfältig das folgende Programm ein, das Fahrenheitgrade in Grad Celsius umrechnet.

```
1Ø REM Temperaturumrechnung
2Ø PRINT "Grad F", "Grad C"
3Ø PRINT
4Ø INPUT "Eingabe Grad F", F
5Ø PRINT F,(F-32)*5/9
6Ø GO TO 4Ø
```

Die Wörter in Zeile 1Ø müssen Sie eintippen. Außerdem: Obwohl **GO TO** (geh zu) einen Zwischenraum hat, ist es in Wirklichkeit ein einziges Schlüsselwort (auf Taste **G**).

Fahren Sie das Programm. Sie werden die Überschriften in Zeile 2Ø auf dem Bildschirm angezeigt sehen, aber was ist aus Zeile 1Ø geworden? Offenbar hat der Computer sie völlig ignoriert. Genau so ist es. **REM** in Zeile 1Ø steht für remark oder reminder (Bemerkung oder Hinweis) und ist allein dazu da, Sie an das zu erinnern, was das Programm leistet. Ein **REM**-Befehl besteht aus **REM**, gefolgt von irgend etwas, das ganz in Ihrem Belieben steht; der Computer beachtet den ganzen Zeileninhalt nicht.

Inzwischen ist der Computer beim **INPUT**-Befehl in Zeile 4Ø und wartet darauf, daß Sie einen Wert für die Variable F eingeben – das können Sie daran erkennen, daß statt des vielleicht erwarteten K-Cursors ein L-Cursor zu sehen ist. Geben Sie eine Zahl ein und vergessen Sie **ENTER** nicht. Der Computer hat nun das Ergebnis angezeigt und wartet auf die nächste Zahl. Das liegt an Zeile 6Ø **GO TO 4Ø**. Sie besagt genau das, was dort steht. Statt daß der Computer das Programm beendet, springt er zurück zu Zeile 4Ø und fängt wieder von vorne an. Sie geben also eine neue Temperatur ein.

Nachdem Sie das ein paarmal ausprobiert haben, werden Sie sich vielleicht fragen, ob den Computer das mit der Zeit nicht langweilt. Tut es nicht. Wenn er die nächste Zahl verlangt, schreiben Sie **STOP**. Der Computer reagiert mit einer Meldung **H STOP in INPUT in line 4Ø:1**, was Ihnen verrät, warum und wo (beim ersten Befehl von 4Ø) er aufgehört hat.

Wenn Sie das Programm fortsetzen wollen, tippen Sie

CONT

und der Computer verlangt von Ihnen wieder eine Zahl.

Bei **CONTINUE** (fahre fort) erinnert der Computer sich an die Zeilennummer in der letzten Meldung, die er Ihnen geschickt hat, solange sie nicht **Ø OK** gelautet hat, und springt zu dieser Zeile zurück. In unserem Fall muß er also zu Zeile 4Ø, dem **INPUT**-Befehl zurückspringen.

Ersetzen Sie Zeile 6Ø durch **GO TO 31** – im Ablauf des Programms wird sich kein merkbarer Unterschied ergeben. Wenn die Zahl in einem **GO TO**-Befehl sich auf eine nicht vorhandene Zeile bezieht, erfolgt der Sprung zur nächsten Zeile nach der eingegebenen Nummer. Das selbe gilt für **RUN**; **RUN** für sich allein bedeutet eigentlich **RUN Ø**.

Geben Sie nun Zahlen ein, bis der Bildschirm voll wird. Wenn er voll ist, schiebt der Computer die ganze obere Bildschirmhälfte eine Zeile hinauf, um Platz zu schaffen, wobei oben die Überschriften wegfallen. Das nennt man Scrolling (Abrollen).

Wenn Sie davon genug haben, halten Sie das Programm mit **STOP** an und holen sich das Listing mit **ENTER**.

Sehen Sie sich die **PRINT**-Anweisung in Zeile 5Ø an. Die Zeichensetzung dort – das Komma (,) – ist sehr wichtig, und Sie sollten sich merken, daß sie sich an viel präzisere Regeln hält als die Interpunktion im Englischen.

Kommas werden verwendet, damit die Anzeige entweder am linken Rand oder in der Mitte des Bildschirms beginnt, je nachdem, was als nächstes an der Reihe ist. So sorgt das Komma in Zeile 5Ø dafür, daß die Temperatur in Celsiusgraden in der Zeilenmitte angezeigt wird. Bei einem Strichpunkt bzw. Semikolon dagegen wird die nächste Zahl oder der nächste String unmittelbar nach dem vorangehenden angezeigt. Sie können das in Zeile 5Ø sehen, wenn das Komma durch ein Semikolon ersetzt wird.

Ein anderes Interpunktionszeichen, das Sie so bei **PRINT** verwenden können, ist der Apostroph ('). Er sorgt dafür, daß das als nächstes Anzuzeigende am Anfang der nächsten Zeile auf dem Bildschirm erscheint, aber das geschieht ohnehin am Ende jedes **PRINT**-Befehls, so daß Sie den Apostroph kaum brauchen werden. Aus diesem Grund fängt der **PRINT**-Befehl in Zeile 5Ø mit der Anzeige jedesmal in einer neuen Zeile an, und deshalb liefert der **PRINT**-Befehl in Zeile 3Ø auch eine leere Zeile.

Wenn Sie das unterbinden wollen, damit nach einem **PRINT**-Befehl der nächste in derselben Zeile weitergeht, können Sie hinter das Ende des ersten ein Komma oder einen Strichpunkt stellen. Damit Sie sehen, wie das geht, ersetzen Sie Zeile 5Ø der Reihe nach jeweils durch

```
5Ø PRINT F,
5Ø PRINT F;
```

und

```
5Ø PRINT F
```

und fahren jede Version. Um noch ein Übriges zu tun, könnten Sie auch noch

```
5Ø PRINT F'
```

ausprobieren.

Die Zeile mit dem Komma zieht alles in zwei Kolonnen auseinander, die mit dem Strichpunkt preßt alles zusammen, die ohne Zusatz räumt jeder Zahl eine eigene Zeile

ebenso ein wie die mit dem Apostroph – der Apostroph liefert eine eigene neue Zeile, sperrt aber die automatische.

Merken Sie sich den Unterschied zwischen Kommas und Strichpunkten bei **PRINT**-Befehlen; verwechseln Sie sie außerdem nicht mit den Doppelpunkten (**:**), die dazu dienen, Befehle in einer einzigen Zeile zu trennen.

Geben Sie jetzt diese zusätzlichen Zeilen ein:

```
100 REM dieses höfliche Programm merkt sich Ihren Namen
110 INPUT n$
120 PRINT "Hallo "; n$; "!"
130 GO TO 110
```

Das ist ein vom vorherigen getrenntes Programm, aber Sie können Sie beide gleichzeitig im Computer behalten. Um das neue zu fahren, geben Sie ein

```
RUN 100
```

Da dieses Programm als Eingabe anstelle einer Zahl einen String hat, zeigt es zwei String-Anführungszeichen – das ist ein Hinweis für Sie und erspart Ihnen in der Regel auch zusätzliche Schreibarbeit. Probieren Sie es mit Ihrem eigenen Namen und allen Spitznamen, die Ihnen einfallen.

Beim nächstenmal erhalten Sie die Anführungszeichen wieder, aber Sie brauchen sie nicht zu verwenden, wenn Sie nicht wollen. Probieren Sie das Folgende: Löschen Sie sie (mit ◆ und zweimal **DELETE**) und schreiben Sie noch einmal

```
n$
```

Der Computer schreibt nun den letzten Namen, den Sie zwischen die Anführungszeichen eingegeben hatten.

Bei der nächsten Eingabe tippen Sie wieder

```
n$
```

löchen aber diesmal die Anführungszeichen nicht. Zu Ihrer Verwirrung hat die Variable n$ jetzt den Wert "n$".

Wenn Sie für Stringeingabe **STOP** verwenden wollen, müssen Sie den Cursor zuerst mit ◆ zum Anfang der Zeile zurücksetzen.

Sehen wir uns noch einmal das **RUN 100** von vorhin an. Damit springt der Computer einfach zu Zeile **100.** Hätten wir dann nicht statt dessen **GO TO 100** schreiben können? In diesem Fall ist es zufällig, so daß die Antwort Ja lautet; ein Unterschied besteht aber doch. **RUN 100** löscht zuerst alle Variablen und den Bildschirm und wirkt danach genau wie **GO TO 100. GO TO 100** löscht überhaupt nichts. Es kann durchaus Gelegenheiten geben, wo Sie ein Programm fahren wollen, ohne Variablen zu löschen; dann wäre **GO TO** notwendig, und **RUN** könnte eine Katastrophe sein: Demnach gewöhnt man sich besser nicht an, automatisch **RUN** zu tippen, um ein Programm zu fahren.

Ein weiterer Unterschied: Sie können **RUN** ohne eine Zeilennummer eingeben, worauf bei der ersten Programmzeile angefangen wird. **GO TO** braucht stets eine Zeilennummer.

Beide Programme haben angehalten, weil Sie in der Eingabezeile **STOP** geschrieben haben; manchmal schreiben Sie – aus Versehen – ein Programm, das Sie nicht stoppen können, und das nicht von selbst aufhört. Tippen Sie

```
200 GO TO 200
RUN 200
```

Das sieht ganz so aus, als würde es ewig laufen, wenn Sie den Stecker nicht herausziehen; da gibt es aber ein weniger drastisches Mittel. Drücken Sie **CAPS SHIFT** zusammen mit der Taste **SPACE**, auf der oben **BREAK** steht. Das Programm kommt zum stehen mit der Meldung **L BREAK into program** (Unterbrechung während des Programms).

Am Ende jeder Anweisung sieht das Programm nach, ob diese Tasten gedrückt worden sind; wenn das der Fall ist, kommt es zum Stillstand. Die **BREAK**-Taste kann auch verwendet werden, wenn Sie gerade dabei sind, den Kassettenrecorder oder den Drucker oder andere Zusatzgeräte zu benutzen, die man an den Computer anschließen kann – nur für den Fall, der Computer erwartet von Ihnen etwas, das Sie nicht tun.

In diesen Fällen erscheint eine andere Meldung: **D BREAK – CONT repeats.** Dann wiederholt **CONTINUE** hier (und auch in den meisten anderen Fällen) die Anweisung, wo das Programm unterbrochen wurde, aber nach der Meldung **L BREAK into program** fährt **CONTINUE** direkt mit der nächsten Anweisung fort, nachdem er vorgesehene Sprünge zugelassen hat.

Fahren Sie das Namensprogramm noch einmal und geben Sie als Input ein

```
n$
```

(nachdem Sie die Anführungszeichen entfernt haben)

n$ ist eine nicht definierte Variable, und Sie erhalten eine Fehlermeldung **2 Variable not found** (Variable nicht gefunden).

Wenn Sie jetzt eintippen

```
LET n$="etwas Bestimmtes"
```

(wofür die eigene Meldung **0 OK, 0:1** erscheint) und

```
CONTINUE
```

werden Sie feststellen, daß Sie n$ ohne Schwierigkeiten für Eingabedaten verwenden können.

In diesem Fall springt **CONTINUE** zum **INPUT**-Befehl in Zeile 110. Die Meldung von der **LET**-Anweisung bleibt unbeachtet, weil dort 'OK' stand. Der Sprung erfolgt zu dem Befehl, der in der ersten Meldung erwähnt war, dem ersten Befehl in Zeile 110. Das soll eine Hilfestellung sein. Wenn ein Programm bei irgendeinem Fehler unterbricht,

können Sie alles Mögliche unternehmen, um ihn zu beheben, anschließend funktioniert **CONTINUE** immer noch.

Wie vorhin schon erwähnt, ist die Meldung **L BREAK into program** eine Besonderheit, weil **CONTINUE** danach den Befehl nicht wiederholt, wo das Programm zum Stillstand gekommen war.

Die automatischen Listings (also diejenigen, die nicht Folge eines **LIST**-Befehls sind, sondern nach Eingabe einer neuen Zeile erscheinen) geben Ihnen vielleicht Rätsel auf. Wenn Sie ein Programm mit 5Ø Zeilen eingeben, das nur aus **REM**-Anweisungen besteht

```
1 REM
2 REM
3 REM
 :  :
 :  :
49 REM
5Ø REM
```

können Sie damit experimentieren.

Das erste, was man sich merken muß, ist, daß die laufende Zeile (mit ▶) stets auf dem Bildschirm erscheint, gewöhnlich in der Nähe der Mitte.

Tippen Sie

LIST (und natürlich **ENTER**)

und wenn gefragt wird **scroll?** (weil der Bildschirm voll ist) drücken Sie **n** für 'Nein'. Der Computer liefert die Fehlermeldung **D BREAK - CONT repeats**, so, als hätten Sie **BREAK** gedrückt. Bei irgendeiner Gelegenheit können Sie feststellen, was geschieht, wenn Sie statt **n** den Buchstaben **y** eingeben; **n**, **SPACE** und **STOP** zählen als Nein, während alles andere als Ja gewertet wird.

Drücken Sie nun erneut **ENTER**, um automatisch ein Listing zu erhalten. Auf dem Bildschirm müßten Sie die Zeilen 1 bis 22 sehen. Schreiben Sie jetzt

23 REM

worauf Sie die Zeilen 2 bis 23 auf dem Schirm erhalten. Tippen Sie

28 REM

und Sie sehen die Zeilen 7 bis 28. In beiden Fällen haben Sie durch Eingabe einer neuen Zeile den Programmcursor bewegt, so daß ein neues Listing erfolgt ist.

Vielleicht kommt Ihnen das ein bißchen willkürlich vor. In Wahrheit wird versucht, Ihnen genau das zu geben, was Sie wollen. Da die Menschen aber unberechenbare Wesen sind, errät der Computer eben nicht immer das Richtige.

Der Computer merkt sich nicht nur jeweils die laufende Zeile, die auf dem Bildschirm erscheinen muß, sondern auch die oberste Zeile auf dem Schirm. Wenn er ein Listing

zu machen versucht, vergleicht er als erstes die oberste mit der laufenden Zeile.

Kommt die oberste Zeile danach, dann hat es keinen Sinn, dort anzufangen, also wird die laufende Zeile als neue Oberzeile verwendet und das Listing aufgestellt.

Abgesehen davon geht er so vor, daß er beim Listing mit der obersten Zeile anfängt und weitermacht, bis er die laufende Zeile aufgeführt hat, wobei er notfalls abrollt. Zuerst stellt er aber eine grobe Berechnung an, um festzustellen, wie lange das dauern würde, und wenn die Antwort lautet: 'Viel zu lang', zieht er die oberste Zeile herunter, so daß sie der laufenden Zeile viel näher ist. Nachdem er also seine Oberzeile errechnet hat, listet er von dort an auf. Ist die laufende Zeile aufgeführt, sobald er das Ende des Programms oder die Bildschirmunterseite erreicht hat, bleibt er stehen. Ansonsten rollt er ab, bis die laufende Zeile auf dem Bildschirm steht, und nimmt für jede zusätzliche Zeile, die er aufführt, die oberste Zeile um eins herunter, so daß die Oberzeile in die Gegend der laufenden Zeile kommt.

Probieren Sie es aus, die laufende Zeile zu verschieben, indem Sie eingeben

Zeilennummer **REM**

LIST verschiebt die laufende Zeile, aber nicht die oberste, so daß aufeinanderfolgende Listings verschieden ausfallen können. Drücken Sie beispielsweise

LIST

um das **LIST**-Listing zu erhalten, und dann wieder auf **ENTER**, damit Zeile 0 die oberste Zeile wird. Auf dem Bildschirm müßten die Zeilen 1 bis 22 stehen. Tippen Sie

LIST 22

was Ihnen die Zeilen 22 bis 43 bringt. Wenn sie wieder auf **ENTER** drücken, erhalten Sie wieder die Zeilen 1 bis 22. Das ist bei kurzen Programmen nützlicher als bei langen.

Verwenden Sie das Programm voller **REM**-Anweisungen, tippen Sie

LIST

und auf **scroll?** ein **n**. Dann geben Sie

CONT

ein. **CONTINUE** ist hier ein bißchen problematisch, weil der untere Bildschirmteil leer wird; mit **BREAK** können Sie den Normalzustand aber wieder herstellen. Der Grund: **LIST** war der erste Befehl in der Zeile, so daß **CONTINUE** diesen wiederholt. Bedauerlicherweise ist der erste Befehl in der Zeile jetzt **CONTINUE** selbst; der Computer sitzt also da und wiederholt unaufhörlich **CONTINUE**, bis Sie eingreifen.

Sie können das abwandeln, indem Sie **LIST** ersetzen durch

: LIST

nach **CONTINUE** bringt der Computer die Meldung **0 OK** weil **CONTINUE** zum zweiten Befehl in der Zeile springt, der als ihr Ende aufgefaßt wird. Probieren Sie jetzt

```
:: LIST
```

Sie sehen nach **CONTINUE** jetzt die Meldung **N Statement lost** (Anweisung verloren), weil **CONTINUE** zum dritten Befehl in der Zeile springt, den es nicht mehr gibt.

Sie haben jetzt die Anweisungen **PRINT**, **LET**, **INPUT**, **RUN**, **LIST**, **GO TO**, **CONTINUE**, **NEW** und **REM** gesehen. Sie können alle entweder als direkte Befehle oder als Programmzeilen verwendet werden – das gilt übrigens für fast alle Befehle in der *BASIC*-Sprache des ZX Spectrum. **RUN, LIST, CONTINUE** und **NEW** sind in der Regel innerhalb eines Programms nicht von großem Nutzen, aber verwenden kann man sie.

Übungen

1. Setzen Sie eine **LIST**-Anweisung so in ein Programm, daß es sich selbst auflistet, sobald Sie es fahren.

2. Schreiben Sie ein Programm für die Eingabe von Preisen, das die fällige Steuer anzeigt (bei 15 Prozent). Fügen Sie **PRINT**-Anweisungen ein, damit der Computer mitteilt, was er tun wird, und mit großer Höflichkeit den Eingabepreis anfordert. Ändern Sie das Programm so ab, daß Sie auch den Steuersatz eingeben können (entweder Null oder künftige Erhöhungen).

3. Schreiben Sie ein Programm, das die laufenden Summen der von Ihnen eingegebenen Zahlen anzeigt. (Vorschlag: Verwenden Sie zwei Variablen mit den Namen **Gesamt** – zu Beginn auf 0 gestellt – und **Einzel.** Geben Sie bei **Einzel** Werte ein, addieren Sie zu **Gesamt**, zeigen Sie beides an und fangen Sie von vorne an.)

4. Was würden **CONTINUE** und **NEW** in einem Programm leisten? Können Sie sich dafür eine Verwendung überhaupt vorstellen?

KAPITEL 3

Entscheidungen

Zusammenfassung:
IF, STOP
=, <, >, <=, >=, <>

Alle Programme, die wir bisher gesehen haben, waren ziemlich leicht vorhersehbar – sie gingen die Befehle der Reihe nach durch und fingen wieder von vorne an. Sehr nutzbringend ist das nicht. In der Praxis wird vom Computer verlangt, daß er Entscheidungen trifft und dementsprechend handelt. Die dafür verwendete Anweisung hat die Form ... **IF** (wenn) etwas wahr oder nicht wahr ist, **THEN** (dann) mach dies oder jenes.

Beispiel: Drücken Sie **NEW**, um das vorherige Programm im Computer zu löschen, tippen Sie das folgende Programm ein und fahren Sie es. (Deutlich zu sehen, daß das ein Spiel für zwei Personen sein soll!)

```
10 REM Zahl erraten
20 INPUT a: CLS
30 INPUT "Welche Zahl?", b
40 IF b=a THEN PRINT "Das ist richtig": STOP
50 IF b<a THEN PRINT "Zu klein, noch einmal"
60 IFB>a THEN PRINT "Zu groß, noch einmal"
70 GO TO 30
```

Sie können erkennen, daß eine **IF**-Anweisung die Form

IF Bedingung **THEN** ...

hat. Die drei Punkte stehen für eine Folge von Befehlen, die auf gewohnte Weise durch Doppelpunkte getrennt sind. Die Bedingung soll entweder als wahr oder als falsch erkannt werden. Wenn sie sich als wahr herausstellt, werden die Anweisungen im Rest der Zeile nach **THEN** ausgeführt, sonst aber übergangen, und das Programm führt den nächsten Befehl aus.

Die einfachsten Bedingungen vergleichen zwei Zahlen oder zwei Strings miteinander. Sie können also prüfen, ob zwei Zahlen gleich sind oder eine größer als die andere, und ob zwei Strings gleich sind oder (grob gesprochen) einer in alphabetischer Reihenfolge vor dem anderen kommt. Benützt werden die Beziehungen =, <, >, <=, >=, und <>.

= bedeutet 'gleich'. Das ist zwar dasselbe Symbol wie das = in einem **LET**-Befehl, wird aber in einem ganz anderen Sinn verwendet.

< (**SYMBOL SHIFT** mit **R**) heißt 'kleiner als', so daß

1<2
-2<-1
-3<1

alle wahr sind, dagegen

1<Ø
Ø<–2

falsch.

Zeile 4Ø vergleicht **a** und **b**. Sind sie gleich, wird das Programm durch den **STOP**-Befehl angehalten. Die Meldung unten auf dem Bildschirm **9 STOP, statement, 3Ø:3** zeigt, daß die dritte Anweisung oder der dritte Befehl in Zeile 3Ø das Programm zum Stillstand gebracht hat.

Zeile 5Ø bestimmt, ob **b** kleiner als **a**, Zeile 6Ø, ob **b** größer als **a** ist. Ist eine dieser Bedingungen wahr, dann wird der entsprechende Hinweis angezeigt, und das Programm geht weiter zu Zeile 7Ø, die den Computer anweist, zu Zeile 3Ø zurückzugehen und von vorne anzufangen.

Der **CLS**-Befehl (clear screen = Bildschirm löschen) in Zeile 2Ø verhindert, daß die zweite Person sieht, was Sie eingegeben haben.

Das > (**SYMBOL SHIFT** mit **T**) bedeutet 'größer als' und ist genau das Gegenteil von <. Sie können sich die Unterscheidung leicht merken, weil das spitze Ende stets auf die Zahl weist, die kleiner sein soll.

<= (**SYMBOL SHIFT** mit **Q** – Sie dürfen das nicht eingeben als <, gefolgt von =) bedeutet 'kleiner als oder gleich'; es ist also wie <, nur eben auch dann wahr, wenn die beiden Zahlen gleich sind. Demnach ist 2<=2 wahr, dagegen 2<2 falsch.

>= (**SYMBOL SHIFT** mit **E**) heißt 'größer als oder gleich' und ähnelt >.

<> (**SYMBOL SHIFT** mit **W**) bedeutet 'nicht gleich', der Gegensatz zu =.

Mathematiker schreiben <=, >=, und <> als ≦, ≧ und ≠. Bei ihnen heißt es auch '2<3<4', wenn '2<3 und 3<4' gemeint ist, aber in BASIC geht das nicht.

Zur Beachtung: In manchen Versionen von BASIC – aber nicht beim ZX Spectrum – kann die **IF**-Anweisung die Form haben

IF Bedingung THEN Zeilennummer

Das bedeutet dasselbe wie

IF Bedingung **THEN GO TO** Zeilennummer

Übungen

1. Probieren Sie dieses Programm aus

1Ø PRINT "x": STOP: PRINT "y"

Wenn Sie es fahren, zeigt es **x** an und beendet mit der Meldung **9 STOP statement, 1Ø:2**. Geben Sie jetzt ein

CONTINUE

Sie erwarten vielleicht, daß dann zur **STOP**-Anweisung zurückgesprungen wird – **CONTINUE** wiederholt in der Regel die in der Meldung genannte Anweisung. Das wäre hier aber nicht sehr zweckdienlich, weil der Computer dann nur wieder anhalten

würde, ohne **y** anzuzeigen. Deshalb ist das so eingerichtet, daß **CONTINUE** nach Meldung 9 zu dem Befehl nach der **STOP**-Anweisung zurückspringt. In unserem Beispiel zeigt der Computer nach **CONTINUE** also **y** an und erreicht das Ende des Programms.

KAPITEL 4

Schleifen

Zusammenfassung:
FOR, NEXT
TO, STEP

Nehmen wir an, Sie wollen fünf Zahlen eingeben und sie addieren. Eine Methode (geben Sie das nicht ein, wenn Sie nicht sehr pflichteifrig sind) sieht so aus:

```
10 LET gesamt=0
20 INPUT a
30 LET gesamt=gesamt+a
40 INPUT a
50 LET gesamt=gesamt+a
60 INPUT a
70 LET gesamt=gesamt+a
80 INPUT a
90 LET gesamt=gesamt+a
100 INPUT a
110 LET gesamt=gesamt+a
120 PRINT gesamt
```

Die richtige Programmierpraxis ist das nicht. Bei fünf Zahlen mag das noch zu bewältigen sein, aber Sie können sich vorstellen, wie mühsam das schon bei zehn Zahlen wird, und hundert Zahlen zu addieren, wäre nahezu unmöglich.

Es gibt einen viel besseren Weg: Man stellt eine Variable auf, die bis 5 zählt, und hält das Programm dann an. Das geht so (diesmal sollten Sie es eingeben):

```
10 LET gesamt=0
20 LET zaehlen=1
30 INPUT a
40 REM zaehlen=wie oft ist a schon eingegeben worden
50 LET gesamt=gesamt+a
60 LET zaehlen=zaehlen+1
70 IF zaehlen<=5 THEN GO TO 30
80 PRINT gesamt
```

Sehen Sie, wie leicht es wäre, Zeile 70 so zu verändern, daß man zehn oder sogar hundert Zahlen addieren könnte?

Diese Zählweise ist so praktisch, daß es zwei eigene Befehle gibt, um das zu erleichtern: den **FOR**-Befehl und den **NEXT**-Befehl. Sie werden stets gemeinsam verwendet. Mit ihnen leistet das eben eingegebene Programm genau dasselbe wie

```
10 LET gesamt=0
20 FOR z=1 TO 5
30 INPUT a
```

```
40 REM z=wie oft ist a schon eingegeben worden
50 LET gesamt=gesamt+a
60 NEXT z
80 PRINT gesamt
```

(Um dieses Programm aus dem vorigen zu erhalten, brauchen Sie nur die Zeilen 20, 40, 60 und 70 zu redigieren. **TO** ist **SYMBOL SHIFT** mit **F**.)

Beachten Sie, daß wir **zählen** zu **z** verändert haben. Die Zählvariable – oder *Steuer*variable – einer **FOR - NEXT**-Schleife muß als Namen einen Einzelbuchstaben haben.

Das Programm bewirkt, daß **z** die Werte 1 (den *Ursprungs*wert), 2, 3, 4 und 5 (das *Limit*) durchläuft und bei jedem einzelnen die Zeilen 30, 40 und 50 ausgeführt werden. Wenn **z** mit seinen fünf Werten fertig ist, wird Zeile 80 ausgeführt.

Eine zusätzliche Feinheit dabei ist, daß die Steuervariable nicht jedesmal um gerade 1 steigen muß: Sie können diesen Wert 1 nach Wunsch verändern, wenn Sie im **FOR**-Befehl einen **STEP**-Teil einfügen. Die allgemeinste Form bei einem **FOR**-Befehl ist

FOR Steuervariable = Ursprungswert **TO** Limit **STEP** Schritt

wo die Steuervariable ein Einzelbuchstabe ist und Ursprungswert, Limit und Schritt alles Dinge, die der Computer als Zahlen behandeln kann – wie die echten Zahlen selbst oder Summen oder die Namen numerischer Variabler. Ersetzen Sie also Zeile 20 im Programm durch

```
20 FOR z=1 TO 5 STEP 3/2
```

dann durchläuft **z** die Werte 1, 2.5 und 4. Beachten Sie, daß Sie sich nicht auf ganze Zahlen zu beschränken brauchen, und außerdem, daß der Steuerwert das Limit nicht ganz genau zu treffen braucht – die Schleife wird fortgeführt, solange er kleiner als das Limit oder ihm gleich ist.

Versuchen Sie es mit diesem Programm, um die Zahlen von 1 bis 10 in umgekehrter Reihenfolge anzuzeigen.

```
10 FOR n=10 TO 1 STEP -1
20 PRINT n
30 NEXT n
```

Wir sagten vorhin, das Programm bleibt in der Schleife, solange die Steuervariable kleiner als das Limit oder ihm gleich ist. Wenn Sie sich ausrechnen, was das in diesem Fall bedeuten würde, werden Sie erkennen, daß dabei Unsinn herauskommt. Die normale Regel muß abgeändert werden: Wenn der Schritt negativ ist, bleibt das Programm in der Schleife, solange die Steuervariable größer als das Limit oder ihm gleich ist.

Sie müssen aufpassen, wenn Sie zwei **FOR** – **NEXT**-Schleifen gemeinsam fahren, eine innerhalb der anderen. Probieren Sie dieses Programm aus, das die Zahlen für einen kompletten Satz von Dominosteinen mit sechs Punkten anzeigt.

```
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;":";n;" ";
40 NEXT n
50 PRINT
60 NEXT m
```

n-Schleife

m-Schleife

Sie sehen, daß die **n**-Schleife sich völlig innerhalb der **m**-Schleife befindet – sie sind *richtig verschachtelt*. Vermeiden muß man, daß zwei **FOR** – **NEXT**-Schleifen sich überlappen, ohne daß eine von beiden sich ganz innerhalb der anderen befindet, etwa wie hier:

```
 5 REM dieses Programm ist falsch
10 FOR m=0 TO 6
20 FOR n=0 TO m
30 PRINT m;":" ";
40 NEXT m
50 PRINT
60 NEXT n
```

m-Schleife

n-Schleife

Zwei **FOR** – **NEXT**-Schleifen müssen entweder eine innerhalb der anderen oder ganz von einander getrennt werden.

Ebenfalls zu vermeiden ist, daß man von außen mitten in eine **FOR** – **NEXT**-Schleife hineinspringt. Die Steuervariable ist nur dann richtig gesetzt, wenn ihre **FOR**-Anweisung ausgeführt wird, und wenn Sie das übersehen, gerät der Computer in Verwirrung. Wahrscheinlich erhalten Sie eine Fehlermeldung **NEXT without FOR** (Next ohne For) oder **variable not found.**

Nichts kann Sie daran hindern, **FOR** und **NEXT** in einem direkten Befehl zu verwenden. Versuchen Sie es mit

```
FOR m=0 TO 10: PRINT m: NEXT m
```

Sie können das manchmal als (etwas künstliche) Methode nutzen, die Beschränkung zu umgehen, daß Sie innerhalb eines Befehls nie **GO TO** irgendwohin schreiben können, weil ein Befehl keine Zeilennummer hat. Beispiel:

```
FOR m=0 TO 1 STEP 0: INPUT a: PRINT a: NEXT m
```

Step Null sorgt hier dafür, daß der Befehl sich ewig wiederholt.

Dergleichen kann aber eigentlich nicht empfohlen werden. Schleicht sich nämlich ein Fehler ein, haben Sie den Befehl verloren und müssen ihn wieder eingeben – und **CONTINUE** funktioniert hier nicht.

Übungen

1. Eine Steuervariable hat nicht bloß einen Namen und einen Wert wie eine gewöhnliche Variable, sondern auch ein Limit, eine Schrittfolge (Step) und einen Hinweis auf die Anweisung nach dem entsprechenden **FOR**-Kommando. Überzeugen Sie sich davon, daß, wenn die **FOR**-Anweisung ausgeführt wird, alle diese Informationen verfügbar sind (wobei Sie den Ausgangswert als den ersten Wert der Variable nehmen), und auch davon, daß diese Information für die **NEXT**-Anweisung ausreicht, damit sie weiß, um wieviel sie den Wert erhöhen, ob sie zurückspringen soll, und wenn ja, wohin.

2. Fahren Sie das dritte Programm oben und tippen Sie dann

PRINT z

Warum ist die Antwort 6 und nicht 5?
(Lösung: Der **NEXT**-Befehl in Zeile 60 wird fünfmal ausgeführt und zu **z** jedesmal 1 dazuaddiert. Beim letztenmal wird **z** 6, worauf der **NEXT**-Befehl beschließt, nicht zurück in die Schleife, sondern weiterzugehen, weil **z** über sein Limit hinaus ist.)
Was geschieht, wenn Sie in Zeile 20 **STEP 2** einsetzen?

3. Verändern Sie das dritte Programm so, daß es, statt automatisch fünf Zahlen zu addieren, von Ihnen die Eingabe verlangt, wie viele Zahlen Sie addiert haben wollen. Was geschieht beim Fahren des Programms, wenn Sie 0 eingeben, also keine Zahlen addiert haben wollen? Warum könnten Sie sich denken, daß das für den Computer Probleme schafft, obwohl klar ist, was Sie meinen? (Der Computer muß eine Suche nach dem Befehl **NEXT z** anstellen, was in der Regel nicht notwendig ist.) Tatsächlich ist schon für alles gesorgt.

4. Verändern Sie in Zeile 10 des vierten Programms oben **100** zu **10** und fahren Sie das Programm. Es wird auf dem Bildschirm die Zahlen von 100 bis 89 anzeigen und unten dann **scroll?** fragen. Damit sollen Sie Gelegenheit bekommen, die Zahlen zu sehen, die dann oben abgerollt werden. Wenn Sie **n, STOP** oder die **BREAK**-Taste drücken, stoppt das Programm mit der Meldung **D BREAK – CONT repeats.** Drücken Sie irgendeine andere Taste, zeigt es weitere 22 Zahlen an und fragt Sie erneut.

5. Löschen Sie Zeile 30 aus dem vierten Programm. Wenn Sie das neue, gekürzte Programm fahren, zeigt es die erste Zahl an und stoppt mit der Meldung **0 OK**. Tippen Sie

NEXT n

geht das Programm einmal durch die Schleife und zeigt die nächste Zahl an.

KAPITEL 5

Subroutinen (Unterprogramme)

Zusammenfassung:
GO SUB, RETURN

Manchmal haben verschiedene Teile eines Programms ganz ähnliche Aufgaben zu bewältigen, und Sie stellen fest, daß Sie dieselben Zeilen zweimal oder noch öfter eingeben müssen. Das ist aber nicht notwendig. Sie können die Zeilen einmal in einer sogenannten Subroutine (oder auch Unterprogramm) eingeben und sie dann überall sonst im Programm verwenden oder rufen, ohne sie noch einmal eintippen zu müssen.

Dazu benutzt man die Anweisungen **GO SUB** (**GO** to **SUB**routine = geh zum Unterprogramm) und **RETURN**.

Das hat die folgende Form:

GO SUB n

wobei n die Zeilennummer der ersten Zeile im Unterprogramm ist. Das ist ganz wie **GO TO** n, nur merkt sich der Computer, wo die **GO SUB**-Anweisung war, so daß er nach Ausführung der Subroutine dorthin zurückkehren kann. Das tut er dadurch, daß er die Zeilennummer und die Nummer der Anweisung in der Zeile (gemeinsam bilden diese die *Returnadresse*) auf den **GO SUB**-Stapel setzt;

RETURN

nimmt die oberste Returnadresse vom **GO SUB**-Stapel ab und geht zur Anweisung danach.

Sehen wir uns als Beispiel das Zahlenrateprogramm noch einmal an. Geben Sie es wie folgt neu ein:

```
10 REM "Ein umgestelltes Ratespiel"
20 INPUT a: CLS
30 INPUT "Errate die Zahl", b
40 IF a=b THEN PRINT "Richtig": STOP
50 IF a<b THEN GO SUB 100
60 IF a>b THEN GO SUB 100
70 GO TO 30
100 PRINT "Noch einmal"
110 RETURN
```

Die **GO TO**-Anweisung in Zeile 70 ist sehr wichtig, weil das Programm sonst in die Subroutine hineinläuft und, wenn die **RETURN**-Anweisung erreicht wird, einen Fehler findet (**7 RETURN without GO SUB** = Return ohne Go Sub).

Hier noch ein, ziemlich albernes Programm, das die Verwendung von **GO SUB** veranschaulicht.

```
100 LET x=10
110 GO SUB 500
120 PRINT s
130 LET x=x+4
140 GO SUB 500
150 PRINT s
160 LET x=x+2
170 GO SUB 500
180 PRINT s
190 STOP
500 LET S=0
510 FOR y=1 TO x
520 LET s=s+y
530 NEXT y
540 RETURN
```

Stellen Sie fest, ob Sie dahinterkommen, was hier geschieht, wenn das Programm gefahren wird. Die Subroutine beginnt mit Zeile 500.

Eine Subroutine kann jederzeit eine andere Subroutine oder auch sich selbst rufen (dann ist sie eine **rekursive** Subroutine), also keine Angst vor mehreren Lagen.

KAPITEL 6

READ, DATA, RESTORE

Zusammenfassung:
READ, DATA, RESTORE (Lies, Daten, Stell wieder her)

In einigen vorherigen Programmen haben wir gesehen, daß man Informationen oder Daten mit der **INPUT**-Anweisung direkt in den Computer eingeben kann. Manchmal wird das sehr mühsam, vor allem dann, wenn bei jedem Lauf des Programms viele Daten sich wiederholen. Mit den Befehlen **READ, DATA** und **RESTORE** können Sie viel Zeit sparen. Beispiel:

```
10 READ a,b,c
20 PRINT a,b,c
30 DATA 10, 20, 30
40 STOP
```

Eine **READ**-Anweisung besteht aus **READ**, gefolgt von einer Liste der Namen von Variablen, getrennt durch Kommas. Das funktioniert ganz ähnlich wie eine **INPUT**-Anweisung, aber: Statt Sie zu veranlassen, daß Sie die Werte für die Variablen eingeben, sieht der Computer die Werte in der **DATA**-Anweisung nach.

Jede **DATA**-Anweisung ist eine Liste von Ausdrücken – numerische oder Stringausdrücke – getrennt durch jeweils ein Komma. In einem Programm können Sie sie einsetzen, wo Sie wollen. Der Computer beachtet sie nämlich nicht, außer, er führt einen **READ**-Befehl aus. Sie müssen sich das so vorstellen: Die Ausdrücke von allen **DATA**-Anweisungen im Programm werden zu einer langen Liste von Ausdrücken aneinandergefügt, der **DATA**-*Liste.* Wenn der Computer das erstemal mit **READ** einen Wert holt, nimmt er den ersten Ausdruck von der **DATA**-Liste, beim nächstenmal den zweiten, und so arbeitet er sich, wenn er aufeinanderfolgenden **READ**-Anweisungen begegnet, durch die **DATA**-Liste hindurch. (Wenn er versucht, über das Ende der **DATA**-Liste hinauszugehen, zeigt er einen Fehler an.)

Beachten Sie: Es ist Zeitverschwendung, **DATA**-Anweisungen in einen direkten Befehl zu setzen, weil **READ** sie nicht findet. **DATA**-Anweisungen müssen im Programm erscheinen.

Sehen wir uns an, wie sie in dem Programm, das Sie eben eingegeben haben, zusammenpassen. Zeile 10 weist den Computer an, in drei Teilen Daten zu lesen und ihnen die Variablen **a, b** und **c** zuzuteilen. Zeile 20 verlangt mit **PRINT**, daß diese Variablen angezeigt werden. Die **DATA**-Anweisung in Zeile 30 gibt die Werte von **a, b** und **c.** Zeile 40 hält das Programm an. Damit Sie sehen, in welcher Reihenfolge das abläuft, verändern Sie Zeile 20 zu:

```
20 PRINT b,c,a
```

Die Information in **DATA** kann Teil einer **FOR** ... **NEXT**-Schleife sein. Schreiben Sie:

```
10 FOR n=1 TO 6
20 READ D
30 DATA 2,4,6,8,10,12
```

```
40 PRINT D
50 NEXT n
60 STOP
```

Wenn dieses Programm mit **RUN** gefahren wird, können Sie die **READ**-Anweisung die **DATA**-Liste durchgehen sehen. **DATA**-Anweisungen können auch Stringvariable enthalten. Beispiel:

```
10 READ d$
20 PRINT "Das Datum ist", d$
30 DATA "1. Juni 1982"
40 STOP
```

Das ist die einfache Methode, aus der **DATA**-Liste Ausdrücke zu holen: Sie fangen vorne an und gehen sie durch, bis Sie am Ende sind. Sie können den Computer aber auch in der **DATA**-Liste herumspringen lassen, und zwar mit der **RESTORE**-Anweisung. Sie besteht aus **RESTORE**, gefolgt von einer Zeilennummer, und veranlaßt folgende **READ**-Anweisungen, ihre Daten von der ersten **DATA**-Anweisung in oder nach der genannten Zeilennummer zu holen. (Sie können die Zeilennummer auch weglassen; das ist dann so, als hätten Sie die Nummer der ersten Zeile im Programm eingegeben.)

Probieren Sie dieses Programm aus:

```
10 READ a,b
20 PRINT a,b
30 RESTORE 10
40 READ x,y,z
50 PRINT x,y,z
60 DATA 1,2,3
70 STOP
```

In diesem Programm stellten die von Zeile 10 verlangten Daten **a**=1 und **b**=2. Der **RESTORE** 10-Befehl setzte die Variablen neu und ließ **x, y** und **z** mit **READ** lesen, beginnend von der ersten Nummer in der **DATA**-Anweisung. Fahren Sie dasselbe Programm ohne Zeile 30, und sehen Sie sich an, was geschieht.

KAPITEL 7

Ausdrücke

Zusammenfassung:
Operationen: +, –, *, /
Rechenausdrücke, wissenschaftliche Schreibweise, Variablennamen.

Sie haben nun schon einige der Arten kennengelernt, nach denen der ZX Spectrum mit Zahlen rechnen kann. Er kann die vier Rechenarten +, –, *, und / ausführen (denken Sie daran, daß * für Multiplikation verwendet wird und / für Division), und den Wert einer Variablen finden, wenn man ihm den Namen mitteilt.

Das Beispiel:

```
LET Steuer=Betrag*15/100
```

liefert nur eine Andeutung der sehr wichtigen Tatsache, daß diese Rechenarten kombiniert werden können. Eine solche Kombination, wie etwa **Betrag*15/100**, wird ein *Ausdruck* genannt: Ein Ausdruck ist also lediglich eine abgekürzte Art, den Computer anzuweisen, verschiedene Berechnungen der Reihe nach auszuführen. In unserem Beispiel bedeutet der Ausdruck **Betrag*15/100**: 'Hol den Wert der Variablen namens "Betrag", multipliziere ihn mit 15 und teile durch 100'.

Wenn Sie das noch nicht getan haben sollten, empfehlen wir Ihnen, daß Sie das Anleitungsheft durchgehen, um zu erfahren, wie der ZX Spectrum mit Zahlen umgeht und in welcher Reihenfolge mathematische Ausdrücke bewertet werden.

Zur Wiederholung:

Multiplikationen und Divisionen werden zuerst ausgeführt. Sie haben eine *höhere Priorität* als Addition und Subtraktion. Im Verhältnis zueinander haben Multiplikation und Division die selbe Priorität. Das bedeutet, daß die Multiplikationen und Divisionen der Reihe nach von links nach rechts ausgeführt werden. Wenn sie alle bewältigt sind, kommen die Additionen und Subtraktionen an die Reihe – auch sie haben untereinander die selbe Priorität, so daß wir sie der Reihe nach von links nach rechts erledigen.

Obwohl Sie eigentlich nur zu wissen brauchen, ob eine Operation eine höhere oder niedrigere Priorität hat als eine andere, geht das beim Computer so: Er hat für die Priorität jeder Operation eine Zahl zwischen 1 und 16: * und / haben die Priorität 8, + und – die Priorität 6.

Diese Reihenfolge beim Rechnen kennt keine Ausnahmen, aber Sie können sie durch die Verwendung von Klammern umgehen: Alles, was in Klammern steht, wird zuerst bewertet und dann wie eine Einzelzahl behandelt.

Ausdrücke sind deshalb nützlich, weil Sie immer dann, wenn der Computer von Ihnen eine Zahl erwartet, ihm einen Ausdruck geben können und er die Lösung ausrechnen wird. Die Ausnahmen von dieser Regel sind so selten, daß sie in jedem Fall ausdrücklich angegeben werden.

Sie können in einem einzigen Ausdruck so viele Strings (oder Stringvariable) aneinanderfügen, wie Sie wollen, und sogar Klammern verwenden.

Wir sollten Ihnen eigentlich sagen, was Sie als Variablennamen benützen können und was nicht. Wie wir schon erwähnt haben, muß der Name einer Stringvariablen ein Einzelbuchstabe, gefolgt von einem **$** sein, der Name der Steuervariablen bei einer

FOR – **NEXT**-Schleife ein einzelner Buchstabe, aber bei den Namen gewöhnlicher numerischer Variabler hat man viel mehr Freiheit. Jeder Buchstabe und jede Ziffer können Verwendung finden, solange das erste Zeichen ein Buchstabe ist. Sie können auch Leerstellen einfügen, damit sich das leichter lesen läßt; als Teil des Namens zählen sie nicht. Außerdem spielt es für den Namen keine Rolle, ob Sie ihn mit großen oder kleinen Buchstaben schreiben.

Hier einige Beispiele für zulässige Variablennamen:

x
Der Name ist so lang daß ich ihn nie wieder hinschreiben kann ohne einen Fehler zu machen
jetzt sind wir sechs (diese beiden letzten Namen gelten als gleich und
jETZtsinDwiRsEchS beziehen sich auf dieselbe Variable)

Die nachfolgenden Namen sind als Variablennamen nicht zulässig:

2ØØ1 (beginnt mit einer Ziffer)
3 Baeren (beginnt mit einer Ziffer)
M*A*S*H (* ist weder Buchstabe noch Ziffer)
Mueller-Remscheid (– ist weder Buchstabe noch Ziffer)

Numerische Ausdrücke können dargestellt werden durch eine Zahl mit Exponent (auch hier verweisen wir auf das einführende Heft). Probieren Sie das Folgende, damit Sie klarsehen:

```
PRINT 2.34eØ
PRINT 2.34e1
PRINT 2.34e2
```

und so weiter bis zu

```
PRINT 2.34e15
```

Sie werden sehen, daß der Computer nach einer Zeit anfängt, die wissenschaftliche Schreibweise zu verwenden. Der Grund: Man kann nicht mehr als vierzehn Zeichen dafür verwenden, eine Zahl zu schreiben. Versuchen Sie es auch mit

```
PRINT 2.34e-1
PRINT 2.34e-2
```

und so weiter.

PRINT liefert von einer Zahl nur acht geltende Ziffern. Probieren Sie

```
PRINT 4294967295, 4294967295-429e7
```

Das beweist, daß der Computer die Ziffern von 4294967295 aufnehmen kann, auch wenn er nicht bereit ist, sie alle auf einmal anzuzeigen.

Der ZX Spectrum verwendet Fließpunktarithmetik. Das heißt, er hält die Ziffern einer Zahl (ihre Mantisse) und die Position des Punktes (des Exponenten) getrennt. Das ist nicht immer exakt, nicht einmal bei ganzen Zahlen. Schreiben Sie

PRINT 1e10+1-1e10,1e10-1e10+1

Zahlen werden mit einer Genauigkeit von rund neuneinhalb Stellen gespeichert, so daß 1e10 zu groß ist, um ganz richtig aufgenommen zu werden. Die Ungenauigkeit (in der Regel um 2) beträgt mehr als 1, also erscheinen die Zahlen 1e10 und 1e10+1 dem Computer gleich.

Wenn Sie ein noch merkwürdigeres Beispiel erleben wollen, schreiben Sie

PRINT 5e9+1-5e9

Hier beträgt die Ungenauigkeit in 5e9 nur rund 1, und die zu addierende 1 wird tatsächlich auf 2 *aufgerundet*. Die Zahlen 5e9+1 und 5e9+2 erscheinen dem Computer gleich.

Die größte ganze Zahl, die völlig korrekt gespeichert werden kann, ist 1 weniger als 32 Zweien miteinander multipliziert (oder 4 294 967 295).

Der String " " ohne Zeichen darin wird *leerer* oder *Null*string genannt. Denken Sie daran, daß Leerräume bedeutsam sind und ein leerer String nicht dasselbe ist wie einer, der nur Leerstellen enthält.

Probieren Sie

PRINT "Haben Sie schon "Vom Winde verweht" gelesen?"

Wenn Sie **ENTER** drücken, erhalten Sie das blinkende Fragezeichen, das anzeigt, daß irgendwo in der Zeile ein Fehler unterlaufen ist. Wenn der Computer die An-Anführungszeichen vor "Vom Winde verweht" findet, geht er davon aus, daß sie das Ende des Strings "Haben Sie schon" bedeuten, und kann dann nicht erkennen, was "Vom Winde verweht" besagt.

Es gibt ein spezielles Mittel, darüber hinwegzukommen. Sobald Sie mitten in einem String ein Stringanführungszeichen schreiben wollen, müssen Sie das doppelt tun, und zwar so:

PRINT "Haben Sie schon ""Vom Winde verweht"" gelesen?"

Wie Sie aus dem, was auf dem Bildschirm angezeigt wird, ersehen können, ist jedes doppelte Anführungszeichen in Wirklichkeit nur einmal vorhanden, man muß es nur zweimal eingeben, damit es erkannt wird.

KAPITEL 8

Strings (Ketten)

Zusammenfassung:
Slicing, Verwendung von **TO**. Beachten Sie, daß diese Schreibweise nicht dem Standard-BASIC entspricht.

Ein Substring besteht aus einigen aufeinanderfolgenden Zeichen des (Gesamt-) Strings. So ist "string" ein Substring von "großer string", aber "g string" und "gro res" sind es nicht.

Es gibt eine Schreibweise für die Beschreibung von Substrings, die *Slicing* (svw. in Scheiben schneiden) genannt wird. Man kann sie für willkürliche Stringausdrücke verwenden. Die allgemeine Form ist

Stringausdruck (Anfang **TO** Ende)

beispielsweise

"abcdef" (2 TO 5) = "bcde"

Wenn Sie den Anfang weglassen, wird von 1 ausgegangen, wenn das Ende weggelassen wird, geht der Substring bis zum Ende des Strings. So ist

"abcdef"(TO 5) = "abcdef"(1 TO 5) = "abcde"
"abcdef"(2 TO) = "abcdef"(2 TO 6) = "bcdef"
"abcdef"(TO) = "abcdef"(1 TO 6) = "abcdef"

(Das letzte können Sie ebensogut **"abcdef"()** schreiben.)

Eine etwas andere Form läßt das **TO** weg und hat nur eine Zahl.

"abcdef"(3) = "abcdef"(3 TO 3) = "c"

Obwohl normalerweise Anfang wie Ende sich auf vorhandene Teile des Strings beziehen müssen, steht eine andere Regel darüber: Wenn der Anfang höher ist als das Ende, ist die Folge der leere String. Demnach erbringt

"abcdef"(5 TO 7)

die Fehlermeldung **3 subscript wrong** (Index falsch), weil der String nur 6 Zeichen enthält und 7 zu hoch ist, aber

"abcdef"(8 TO 7) = ""

und

"abcdef"(1 TO Ø) = ""

Anfang und Ende dürfen nicht negativ sein, sonst erhalten Sie die Fehlermeldung **B Integer out of range** (Zahl außerhalb des zulässigen Bereichs).

Das nächste Programm ist ganz einfach und veranschaulicht einige dieser Regeln.

```
10 LET a$="abcdef"
20 FOR n=1 TO 6
30 PRINT a$ (n TO 6)
40 NEXT n
50 STOP
```

Drücken Sie **NEW**, wenn dieses Programm abgelaufen ist, und geben Sie das nächste ein:

```
10 LET a$="ABLE WAS I"
20 FOR n=1 TO 10
30 PRINT a$(n TO 10),a$((10-n) TO 10)
40 NEXT n
50 STOP
```

Bei Stringvariablen können wir nicht nur Substrings entnehmen, sondern ihnen auch etwas zuteilen. Beispiel:

```
LET a$="Ich bin ZX Spectrum"
```

und dann

```
LET a$(5 TO 8)="******"
```

und

```
PRINT a$
```

Beachten Sie, daß nur die ersten vier Sternchen verwendet worden sind, weil der Substring **a$(5 TO 8)** nur 4 Zeichen lang ist. Das ist eine typische Eigenschaft der Ergänzung von Substrings.

Der Substring muß hinterher genau dieselbe Länge haben wie vorher. Um sicherzustellen, daß das auch geschieht, wird der String, dem man etwas zuteilt, rechts abgeschnitten, wenn er zu lang, oder mit Leerräumen aufgefüllt, wenn er zu kurz ist. Man nennt das Prokrustesbett, nach dem Gastwirt Prokrustes, der dadurch dafür sorgte, daß seine Gäste ins Bett paßten, daß er sie entweder mit der Streckfolter längte oder ihnen die Füße abhackte.

Versuchen Sie es jetzt mit

```
LET a$()="Hallo Leute"
```

und

```
PRINT a$;"."
```

dann sehen Sie, daß wieder dasselbe eingetreten ist (diesmal mit eingesetzten Leerräumen), weil **a$()** als Substring zählt. Richtig geht es mit

LET a$="Hallo Leute"

Komplizierte Stringausdrücke müssen in Klammern gestellt werden, bevor man sie zerschneiden kann. Beispiel:

"abc" + "def"(1 TO 2)="abcde"
("abc" + "def") (1 TO 2)="ab"

Übung:
1. Versuchen Sie ein Programm zu schreiben, das mit String slicing den Wochentag anzeigt. Ein Tip: Der String kann heißen SonMonDieMitDonFreSam oder auch SOMODIMIDOFRSA.

KAPITEL 9

Funktionen

Zusammenfassung:
DEF
LEN, STR$, VAL, SGN, ABS, INT, SQR
FN

Denken Sie sich eine Wurstmaschine. Sie stecken an einem Ende ein Stück Fleisch hinein, drehen eine Kurbel, und am anderen Ende kommt die Wurst heraus. Bei Schweinefleisch gibt es Schweinswurst, bei Fisch eine Fischwurst, bei Rindfleisch eine Rindfleischwurst.

Funktionen verarbeiten ganz ähnlich Zahlen und Strings. Sie liefern einen Wert (*Argument* genannt), zerkleinern ihn, indem Sie Berechnungen damit anstellen, und erhalten schließlich einen anderen Wert, das *Resultat*.

Fleisch hinein → Wurstmaschine → Wurst heraus

Argument hinein → Funktion → Resultat heraus

Verschiedene Argumente führen zu verschiedenen Resultaten. Ist das Argument völlig unangemessen, kommt die Funktion zum Stillstand und liefert eine Fehlermeldung.

Genau so, wie man verschiedene Maschinen haben kann, um verschiedene Produkte herzustellen – eine für Würste, eine zweite für Geschirrtücher, eine dritte für Fischstäbchen, usw., bewältigen verschiedene Funktionen verschiedene Rechenaufgaben. Jede besitzt ihren eigenen Wert, die sie von den anderen unterscheidet.

So verwenden Sie in den Ausdrücken eine Funktion: Sie geben den Namen der Funktion und das Argument ein. Das Resultat der Funktion wird berechnet, sobald der Ausdruck zur Behandlung kommt.

Beispiel: Die Funktion **LEN**. Sie errechnet die Länge eines Strings. Ihr Argument ist der String, dessen Länge Sie feststellen wollen, ihr Resultat die Länge. Wenn Sie also schreiben

```
PRINT LEN "Sinclair"
```

zeigt der Computer die Antwort 8 an, nämlich die Zahl der Buchstaben im Wort 'Sinclair'. (Um **LEN** ebenso wie die meisten Funktionsnamen zu erreichen, müssen Sie in den erweiterten Modus gehen: Drücken Sie gleichzeitig **CAPS SHIFT** und **SYMBOL SHIFT**, damit aus dem **L**-Cursor ein **E**-Cursor wird, und dann die Taste **K**.)

Wenn Sie Funktionen und Operationen in einem einzigen Ausdruck mischen, werden die Funktionen vor den Operationen errechnet. Auch hier können Sie die Regel dadurch umgehen, daß Sie Klammern verwenden. Beispiel: Hier zwei Ausdrücke, die sich nur durch die Klammern unterscheiden. Trotzdem werden die Berechnungen jeweils in völlig unterschiedlicher Reihenfolge ausgeführt (auch wenn am Ende zufällig dasselbe herauskommt).

```
LEN "Karl" + LEN "Knacke"          LEN ("Karl" + "Knacke")
4+LEN "Knacke"                     LEN ("KarlKnacke")
      4+6                          LEN "KarlKnacke"
       10                                  10
```

Jetzt zu anderen Funktionen:

STR$** verwandelt Zahlen in Strings. Das Argument dieser Funktion ist eine Zahl, ihr Resultat der String, der auf dem Bildschirm erscheinen würde, wenn die Zahl durch eine **PRINT**-Anweisung angezeigt werden würde. Beachten Sie, daß ihr Name mit einem **$-Zeichen endet, um anzuzeigen, daß ihr Resultat ein String ist. Beispielsweise könnten Sie sagen

```
LET a$=STR$ 1e2
```

was genau dieselbe Wirkung hätte wie

```
LET a$="100"
```

Oder Sie könnten sagen

```
PRINT LEN STR$ 100.0000
```

und die Antwort 3 erhalten, weil **STR$ 100.0000="100".**

VAL ist wie **STR$**, nur umgekehrt: Die Funktion verwandelt Strings in Zahlen. Beispiel:

```
VAL "3.5"=3.5
```

In gewissem Sinn ist **VAL** die Umkehrung von **STR$**, denn wenn Sie irgendeine Zahl nehmen, zuerst **STR$** und dann **VAL** darauf anwenden, erhalten Sie wieder die Zahl, mit der Sie angefangen haben.

Nehmen Sie aber einen String, wenden zuerst **VAL** und dann **STR$** darauf an, erhalten Sie Ihren ersten String nicht immer wieder.

VAL ist eine äußerst leistungsfähige Funktion, weil der String, der ihr Argument darstellt, nicht darauf beschränkt ist, wie eine einfache Zahl auszusehen – in Frage kommt vielmehr jeder numerische Ausdruck. So, zum Beispiel

```
VAL "2*3"= 6
```

oder sogar

```
VAL ("2"+"*3") = 6
```

Hier sind zwei Vorgänge am Werk. Beim ersten wird das Argument von **VAL** als ein String behandelt: Der Stringausdruck **"2"+"*3"** wird verwertet, um den String **"2*3"** zu

liefern. Dann werden dem String die doppelten Anführungsstriche genommen, der Rest wird als eine Zahl behandelt, so daß **2*3** die Zahl **6** ergibt.

Das kann recht verwirrend werden, wenn man nicht aufpaßt. Beispiel:

PRINT VAL "VAL""VAL""""2""""""""""""

(Denken Sie daran, daß innerhalb eines Strings ein String-Anführungszeichen doppelt geschrieben werden muß. Wenn Sie sich noch tiefer auf Strings einlassen, werden Sie dahinterkommen, daß Stringanführungszeichen vervierfacht oder sogar verachtfacht werden müssen.)

Es gibt noch eine weitere, **VAL** sehr ähnliche Funktion, die aber vielleicht weniger nützlich ist und den Namen **VAL$** hat. Auch ihr Argument ist ein String, ihr Resultat aber ebenfalls einer. Um zu sehen, wie das funktioniert, erinnern Sie sich daran, wie **VAL** in zwei Schritten vorgeht: Zuerst wird ihr Argument als ein String behandelt, dann werden die Stringanführungszeichen entfernt, während der Rest als eine Zahl behandelt wird. Bei **VAL$** ist der erste Schritt derselbe, nachdem aber beim zweiten Schritt die Anführungszeichen entfernt worden sind, wird auch der Rest als String behandelt. Demgemäß ist

VAL$ """Kalte Bowle""" = "Kalte Bowle"

(Beachten Sie, wie auch hier wieder die Anführungszeichen häufig werden.) Schreiben Sie

LET a$="99"

und lassen Sie alles Folgende anzeigen: **VAL a$, VAL "a$", VAL """a$""", VAL$ a$, VAL$ "a$"** und **VAL$ """a$"""**. Manches davon geht, anderes nicht. Versuchen Sie das zu erklären. (Kühlen Kopf bewahren.)

SGN ist die *Vorzeichen*-Funktion (manchmal auch **Signum** genannt). Das ist für Sie die erste Funktion, die nichts mit Strings zu tun hat, weil sowohl ihr Argument als auch ihr Resultat Zahlen sind. Das Resultat ist +1, wenn das Argument positiv, 0, wenn das Argument Null, und –1, wenn das Argument negativ ist.

ABS ist eine weitere Funktion, deren Argument und Resultat beide Male eine Zahl ist. Sie verwandelt das Argument in eine positive Zahl (die das Resultat ist), indem sie das Vorzeichen vergißt. Beispiel:

ABS –3.2 = ABS 3.2 = 3.2

INT heißt 'integer part'. Integer ist eine ganze Zahl, die auch negativ sein kann. Diese Funktion verwandelt eine Bruchzahl dadurch in eine ganze Zahl, daß sie den Bruchteil fallen läßt. Beispiel:

INT 3.9 = 3

Vorsicht beim Umgang mit negativen Zahlen, denn die Funktion rundet immer ab.

Beispiel:

INT –3.9 = –4

SQR berechnet die *Quadratwurzel* einer Zahl – das Resultat, das, mit sich selbst multipliziert, das Argument liefert. Beispiel:

SQR 4 = 2 weil 2*2=4
SQR Ø.25 = Ø.5 weil Ø.5*Ø.5=Ø.25
SQR 2 = 1.4142136 (angenähert), weil
1.4142136*1.4142136=2.ØØØØØØ1

Wenn Sie irgendeine Zahl (auch eine negative) mit sich selbst multiplizieren, ist das Ergebnis immer positiv. Das heißt: Negative Zahlen haben keine Quadratwurzeln. Wenn Sie **SQR** also auf ein negatives Argument anwenden, erhalten Sie die Fehlermeldung **An Invalid Argument** (ungültiges Argument).

Sie können auch eigene Funktionen definieren. Mögliche Namen dafür sind **FN**, gefolgt von einem Buchstaben (wenn das Resultat eine Zahl) oder **FN** mit einem Buchstaben gefolgt von **$** (wenn das Resultat ein String ist). Diese Funktionen sind bei Klammern viel strenger: Das Argument muß in Klammern stehen.

Eine Funktion definieren Sie dadurch, daß Sie irgendwo im Programm eine **DEF**-Anweisung unterbringen. Beispiel: Hier die Definition einer Funktion **FN s**, deren Resultat das Quadrat des folgenden Arguments ist:

```
1Ø DEF FN s(x)=x*x: REM das Quadrat von x
```

DEF ist erhältlich in erweitertem Modus, mit **SYMBOL SHIFT** und **1**. Wenn Sie das eingeben, liefert Ihnen der Computer automatisch **FN**, weil in einer **DEF**-Anweisung dem **DEF** grundsätzlich stets das **FN** folgt. Anschließend vervollständigt das **s** den Namen **FN s** der Funktion.

Das **x** in Klammern ist ein Name, mit dem Sie sich auf das Argument der Funktion beziehen wollen. Sie können dafür jeden Einzelbuchstaben verwenden (oder, wenn das Argument ein String ist, einen Einzelbuchstaben, gefolgt von **$**).

Nach dem = Zeichen kommt die eigentliche Definition der Funktion. Das kann jeder beliebige Ausdruck sein und sich auch mit Verwendung des Namens, den Sie ihm gegeben haben, auf das Argument beziehen (in diesem Fall **x**), als wäre es eine ganz gewöhnliche Variable.

Wenn Sie diese Zeile eingegeben haben, können Sie die Funktion genauso aufrufen wie irgendeine Funktion des Computers, indem Sie ihren Namen **FN s** und danach das Argument eingeben. Nicht vergessen: Wenn Sie selbst eine Funktion bestimmt haben, muß das Argument in Klammern stehen. Versuchen Sie es ein paarmal:

```
PRINT FN s(2)
PRINT FN s(3+4)
PRINT 1+INT FN s (LEN "Elefant"/2+3)
```

Sobald Sie die entsprechende **DEF**-Anweisung ins Programm gesetzt haben, können Sie Ihre eigenen Funktionen in Ausdrücken genauso frei verwenden wie die des Computers.

Bei manchen BASIC-Dialekten müssen Sie sogar das Argument einer der Computerfunktionen in Klammern setzen. Beim ZX Spectrum-BASIC ist das nicht erforderlich.

INT rundet immer ab. Um zur nächsten ganzen Zahl zu runden, addieren Sie zuerst .5 – Sie könnten Ihre eigene Funktion schreiben, die das bewirkt.

```
20 DEF FN r(x)=INT(x+.5): REM liefert x gerundet zur nächsten
   ganzen Zahl
```

Sie erhalten dann beispielsweise

```
FN r(2.9) = 3        FN r(2.4) = 2
FN r(-2.9) = -3      FN r(-2.4) = -2
```

Vergleichen Sie das mit den Antworten, die Sie erhalten, wenn Sie **INT** statt **FN r** verwenden. Geben Sie das folgende Programm ein und fahren Sie es:

```
10 LET x=0: LET y=0: LET a=10
20 DEF FN p(x,y)=a+x*y
30 DEF FN q()=a+x*y
40 PRINT FN p(2,3), FN q()
```

In diesem Programm stecken allerlei Feinheiten.

Erstens ist eine Funktion nicht auf nur ein Argument beschränkt. Sie kann mehrere haben oder auch gar keine – aber die Klammern brauchen Sie immer.

Zweitens spielt es keine Rolle, wo im Programm Sie die **DEF**-Anweisungen unterbringen. Nachdem der Computer Zeile 10 ausgeführt hat, überspringt er die Zeilen 20 und 30 einfach, um zu Zeile 40 zu gelangen. Irgendwo im Programm müssen sie aber stehen. In einem Befehl dürfen sie nicht erscheinen.

Drittens: **x** und **y** sind sowohl die Namen von Variablen im Programm als Ganzem als auch die Namen von Argumenten für die Funktion **FN p. FN p** vergißt die Variablen **x** und **y** vorübergehend, da sie aber kein Argument namens **a** hat, merkt sie sich weiter die Variable **a**. Sobald **FN p(2,3)** behandelt wird, hat **a** damit den Wert 10, weil es die Variable ist, **x** hat den Wert 2, weil es das erste Argument ist, und **y** hat den Wert 3, weil es das zweite Argument ist. Das Resultat ist somit 10+2*3=16. Wird dagegen **FN q()** behandelt, gibt es keine Argumente, deshalb beziehen **a, x** und **y** sich alle auf die Variablen und haben der Reihe nach die Werte 10, 0 und 0. Die Lösung ist in diesem Fall 10+0*0=10.

Verändern Sie nun Zeile 20 zu

```
20 DEF FN  p(x,y)=FN q()
```

Diesmal hat **FN p(2,3)** den Wert 10, weil **FN q** trotzdem zu den Variablen **x** und **y** zurückgeht, statt die Argumente von **FN p** zu verwenden.

Manche BASIC-Versionen (nicht das BASIC beim ZX Spectrum) haben Funktionen mit den Namen LEFT$, RIGHT$, MID$ und TL$.

LEFT$ (a$,n) liefert den Substring von a$, der aus den ersten *n* Zeichen besteht (left = links).

RIGHT$ (a$,n) liefert den Substring von a$, der aus den Zeichen vom *n*-ten an besteht (right = rechts).

MID$ (a$, n1, n2) liefert den Substring von a$, bestehend aus *n*2 Zeichen, beginnend beim *n*-ersten (mid = Mitte).

TL$ (a$) liefert den Substring von a$, der aus allen Zeichen außer dem ersten besteht.

Sie können mit dem ZX Spectrum leicht Funktionen schreiben, die dasselbe bewirken. Beispiel:

```
10 DEF FN t$(a$)=a$(2 TO): REM TL$
20 DEF FN l$(a$,n)=a$( TO n): REM LEFT$
```

Prüfen Sie nach, daß diese Funktionen bei Strings der Länge 0 oder 1 wirksam sind.

Beachten Sie, daß **FN l$** zwei Argumente hat, eine Zahl und einen String. Eine Funktion kann bis zu 26 numerische Argumente haben (warum gerade 26?) und gleichzeitig bis zu 26 Stringargumente.

Übung

Verwenden Sie die Funktion **FN s(x)=x*x** dazu, um **SQR** zu prüfen. Sie sollten feststellen, daß

```
FN s(SQR x)=x
```

wenn Sie jede beliebige positive Zahl für **x** einsetzen, und

```
SQR FN s(x)=ABS x
```

egal, ob **x** positiv ist oder negativ (Warum das **ABS**?).

KAPITEL 10

Mathematische Funktionen

Zusammenfassung:
PI, EXP, LN, SIN, COS, TAN, ASN, ACS, ATN

Dieses Kapitel befaßt sich mit der Mathematik, die der ZX Spectrum bewältigen kann. Es ist durchaus denkbar, daß Sie davon nie etwas gebrauchen können. Wenn Ihnen das also zu schwer erscheinen sollte, dürfen Sie es ruhig überspringen. Es behandelt die Operation ↑ (zur Potenz erheben), die Funktionen **EXP** und **LN** und die trigonometrischen Funktionen **SIN, COS, TAN** und ihre Umkehrungen **ASN, ACS** und **ATN.**

↑ und **EXP**
Sie können eine Zahl zur Potenz einer anderen erheben – das heißt 'multipliziere die erste Zahl mit sich selbst so oft, wie die zweite Zahl es angibt'. Das wird normalerweise dadurch angezeigt, daß man die zweite Zahl rechts neben der ersten hochstellt. Da das bei einem Computer aber schwierig wäre, verwenden wir stattdessen das Symbol ↑. Beispielsweise sind die Potenzen von 2

2 ↑ 1=2
2 ↑ 2=2*2=4 (2 im Quadrat, sonst 2^2 geschrieben)
2 ↑ 3=2*2*2=8 (2 kubiert, sonst 2^3 geschrieben)
2 ↑ 4=2*2*2*2=16 (2 hoch vier, sonst 2^4 geschrieben)

So bedeutet auf einfachster Stufe *'a ↑ b'* = *'a* wird mit sich selbst *b*-mal multipliziert', aber das ergibt offenkundig nur dann Sinn, wenn *b* eine positive ganze Zahl ist. Um eine Definition zu finden, die auch bei anderen Werten von *b* gültig ist, befassen wir uns mit der Regel

a ↑ (b+c)=a ↑ b*a ↑ c

(Beachten Sie, daß wir ↑ eine höhere Priorität geben als * und /, so daß dort, wo mehrere Operationen in einem Ausdruck vorkommen, diese vor den * und / ausgeführt werden.) Es sollte keine große Überzeugungskraft brauchen, Ihnen klarzumachen, daß das funktioniert, wenn *b* und *c* beide positive ganze Zahlen sind. Wenn das aber auch dann gültig sein soll, wo sie es nicht sind, sehen wir uns gezwungen, zu akzeptieren, daß

a ↑ 0=1
a ↑ (–b)=1/a ↑ b
a ↑ (1/b)=b-te Wurzel von *a*, also die Zahl, die Sie *b*-mal mit sich selbst multiplizieren müssen, um *a* zu erhalten

und
a ↑ (b*c)=(a ↑ b) ↑ c

Wenn Sie so etwas zum erstenmal sehen, dann versuchen Sie nicht, sich das gleich alles zu merken. Denken Sie nur daran, daß

a ↑ (–1) = 1/a
und
a ↑ (1/2) = **SQR** a

Wenn Sie sich damit vertraut gemacht haben, ergibt das Übrige mit der Zeit vielleicht auch Sinn.

Experimentieren Sie mit diesen Dingen und probieren Sie dazu dieses Programm aus:

```
10 INPUT a,b,c
20 PRINT a ↑ (b+c),a ↑ b*a ↑ c
30 GO TO 10
```

Wenn die Regel, die wir vorhin genannt haben, richtig ist, werden die beiden Zahlen, die der Computer nach jeder Runde anzeigt, natürlich gleich sein. (Beachten Sie: Wegen der Art und Weise, wie der Computer ↑ berechnet, darf die Zahl auf der linken Seite – hier **a** – niemals negativ sein.)

Ein recht typisches Beispiel für die Nutzung dieser Funktion ist die Zinseszinsberechnung. Nehmen wir an, Sie haben Ihr Geld einer Baugesellschaft geliehen und erhalten im Jahr dafür 15% Zinsen. Nach einem Jahr haben Sie nicht nur die 100%, die Sie vorher schon hatten, sondern auch die 15% Zinsen, die Sie von der Baugesellschaft erhalten, was zusammen 115% des Anfangsbetrages ausmacht. Anders ausgedrückt: Sie haben Ihren Geldbetrag mit 1.15 multipliziert, gleichgültig, wie hoch der Betrag vorher gewesen sein mag. Nach einem zweiten Jahr ist dasselbe noch einmal geschehen. Sie haben dann 1.15*1.15 = 1.15 ↑ 2 = 1.3225mal Ihren ursprünglichen Geldbetrag. Nach **y** Jahren haben Sie, ganz allgemein gesprochen, **1.15** ↑ **y** mal das, womit Sie angefangen haben.

Wenn Sie diesen Befehl ausprobieren

```
FOR y=0 TO 100: PRINT y,10*1.15 ↑ y: NEXT y
```

werden Sie sehen, daß das sehr schnell anwächst, selbst wenn Sie nur mit einem kleinen Betrag angefangen haben, und zwar mit der Zeit immer rapider. (Trotzdem werden Sie vielleicht feststellen, daß die Inflation damit auch noch nicht ausgeglichen wird.)

Diese Art von Verhalten, wo nach einer festen Zeitspanne eine Größe sich um einen festen Anteil erhöht, nennt man *geometrische Progression*. Sie wird dadurch berechnet, daß man eine feste Zahl in die Potenz der Zeit erhebt.

Angenommen, Sie machen Folgendes:

```
10 DEF FN a(x)=a ↑ x
```

Hier ist a durch **LET**-Anweisungen mehr oder weniger festgelegt. Sein Wert entspricht dem Zinssatz, der sich nur in längeren Zeitabständen verändert.

Es gibt für **a** einen bestimmten Wert, der die Funktion **FN a** für das geübte Auge eines Mathematikers besonders angenehm erscheinen läßt. Dieser Wert wird e genannt. Der ZX Spectrum hat eine Funktion namens **EXP**, definiert durch

EXP x = e ↑ x

Leider ist e für sich genommen keine besonders schöne Zahl, sondern eine unendliche, nicht-periodische Dezimale. Ihre ersten Dezimalstellen können Sie sehen, wenn Sie eingeben

PRINT EXP 1

weil **EXP 1 = e ↑ 1 = e**. Das ist natürlich nur eine Annäherung. Man kann **e** nie exakt festlegen.

LN

Die Umkehrung einer Exponentialfunktion ist eine *logarithmische* Funktion: Der *Logarithmus* (bei einer Basis a) einer Zahl *x* ist die Potenz, zu der Sie *a* erheben müssen, um die Zahl *x* zu erhalten, und wird $\log_a x$ geschrieben. Der Definition nach also $a \uparrow \log_a x = x$, und ebenso $\log (a \uparrow x) = x$.

Sie wissen vielleicht schon, wie man Zehnerlogarithmen für Multiplikationen verwendet; man nennt sie *gewöhnliche* Logarithmen. Der ZX Spectrum besitzt eine Funktion **LN**, die Logarithmen der Basis e berechnet; diese werden *natürliche* Logarithmen genannt. Um Logarithmen irgendeiner anderen Basis zu berechnen, müssen Sie den natürlichen Logarithmus durch den natürlichen Logarithmus der Basis teilen:

$\log_a x =$ **LN x/ LN a**

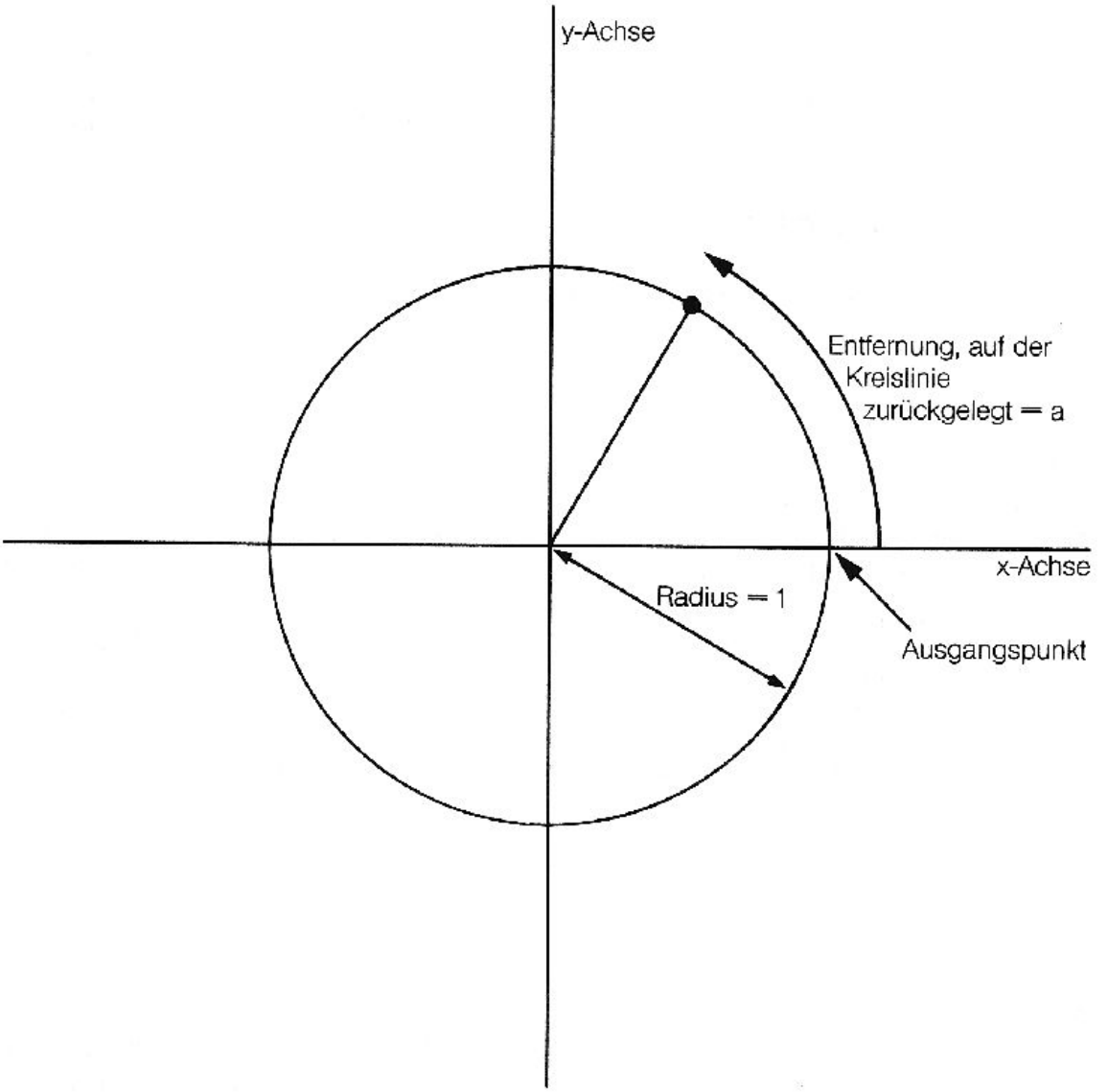
PI

Für jeden Kreis können Sie den Umfang bestimmen, wenn Sie den Durchmesser mit einer Zahl π multiplizieren. (π ist das griechische p und wird verwendet, weil es für Peripherie, also die äußere Umgrenzungslinie, steht, gesprochen pi.)

Wie e ist π ein unendliches, nicht-periodisches Dezimal; es beginnt als 3.141592653589... Das Wort **PI** auf dem Spectrum (erweiterter Modus, dann *M*) wird als diese Zahl verstanden – versuchen Sie **PRINT PI**.

SIN, COS und TAN; ASN, ACS und ATN

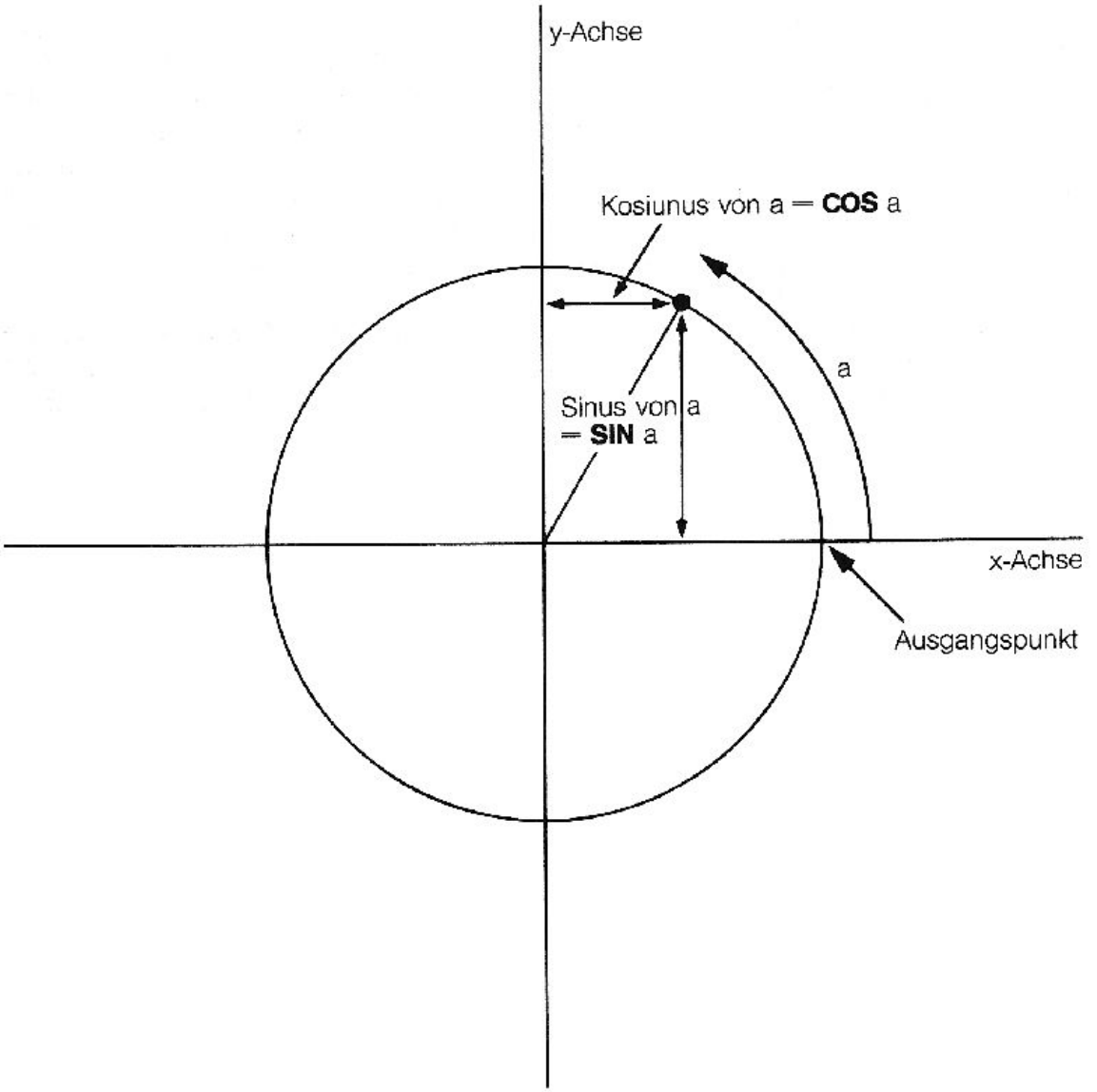
Die *trigonometrischen* Funktionen messen, was vorgeht, wenn ein Punkt sich auf einer Kreislinie bewegt. Unten ein Kreis vom Radius 1 (1 was? Das spielt keine Rolle, solange wir durchgehend bei derselben Einheit bleiben. Nichts kann Sie hindern, für jeden Kreis, der Sie zufällig interessiert, selbst eine neue Einheit zu erfinden) und ein Punkt, der sich rundherum bewegt. Der Punkt begann bei der Stellung 3 Uhr und bewegte sich dann im entgegengesetzten Uhrzeigersinn.



Außerdem haben wir zwei Linien durch die Kreismitte gezeichnet, die *Achsen* genannt werden. Die durch 9 Uhr und 3 Uhr gehende heißt *x-Achse*, die durch 6 Uhr und 12 Uhr *y-Achse*.

Um zu bestimmen, wo sich der Punkt befindet, gibt man an, wie weit er sich vom Ausgangspunkt bei 3 Uhr entfernt hat. Nennen wir diese Entfernung *a*. Wir wissen, daß der Umfang des Kreises $2*\pi$ beträgt (weil sein Radius 1 ist und der Durchmesser damit 2). Wenn er sich also ein Viertel des Weges im Kreis bewegt hat, $a=\pi/2$, beim halben $a=\pi$, und beim ganzen $a=2\pi$.

Wenn Sie den gekrümmten Weg auf der Außenlinie *a* kennen, sind zwei andere Entfernungen, die Sie vielleicht kennen wollen, wie weit der Punkt sich rechts von der y-Achse, und wie weit er sich über der x-Achse befindet. Diese heißen der *Cosinus*, beziehungsweise der *Sinus* von *a*. Die Funktionen **COS** und **SIN** auf dem Computer berechnen sie.



Beachten Sie: Wenn der Punkt links über die y-Achse hinausgeht, wird der Cosinus negativ; geht er unter die x-Achse, wird der Sinus negativ.

Eine weitere Eigenschaft: Sobald *a* 2π erreicht hat, befindet sich der Punkt wieder dort, wo er angefangen hat, und Sinus und Cosinus nehmen wieder dieselben Werte an:

SIN (a+2*PI) = SIN a
COS (a+2*PI) = COS a

Die *Tangente* von a wird definiert als Sinus, geteilt durch den Cosinus; die entsprechende Funktion am Computer heißt **TAN.**

Manchmal müssen wir diese Funktionen umgekehrt berechnen und den Wert von a finden, der Sinus, Cosinus oder Tangente geliefert hat. Die dazu erforderlichen Funktionen werden *Arkussinus* (am Computer **ASN**), Arkuscosinus (**ACS**) und *Arkustangens* (**ATN**) genannt.

Sehen Sie sich im Diagramm des Punktes, der im Kreis läuft, den Radius (Halbmesser) an, der den Mittelpunkt mit dem Punkt verbindet. Sie sollten erkennen können, daß die Entfernung, die wir *a* genannt haben, die Entfernung also, die der Punkt auf der Kreislinie zurückgelegt hat, ein Mittel ist, den Winkel zu messen, durch den der Radius sich von der x-Achse entfernt hat. Wenn $a=\pi/2$, beträgt der Winkel 90 Grad, wenn $a=\pi$, dann 180 Grad, und so weiter bis $a=2\pi$, wo der Winkel 360 Grad mißt. Die Gradeinteilung können Sie auch vergessen und den Winkel allein nach *a* bestimmen. Wir sagen dann, daß wir den Winkel in *Radianten* messen. So sind $\pi/2$ Radianten = 90 Grad, und so weiter.

Sie müssen sich stets merken, daß beim ZX Spectrum **SIN, COS** usw. *Radianten* und nicht Grad verwenden. Um Grad in Radianten zu verwandeln, teilen Sie durch 180 und multiplizieren mit π; umgekehrt teilen Sie durch π und multiplizieren mit 180.

KAPITEL 11

Zufallszahlen

Zusammenfassung:
RANDOMIZE
RND

Dieses Kapitel behandelt die Funktion **RND** und das Schlüsselwort **RANDOMIZE**. Sie werden beide in Verbindung mit Zufallszahlen verwendet, also müssen Sie darauf achten, sie nicht zu verwechseln. Sie stehen beide auf derselben Taste (T); **RANDOMIZE** mußte abgekürzt werden zu **RAND**.

In mancher Beziehung ist **RND** wie eine Funktion. Es führt Berechnungen aus und liefert ein Resultat. Aus dem Rahmen fällt es dadurch, daß es kein Argument benötigt.

Jedesmal, wenn Sie es verwenden, ist sein Resultat eine neue Zufallszahl zwischen Ø und 1. (Manchmal kann es den Wert Ø annehmen, aber niemals 1.)

Probieren Sie

```
1Ø PRINT RND
2Ø GO TO 1Ø
```

um sich anzusehen, wie unterschiedlich die Ergebnisse ausfallen. Können Sie ein Schema erkennen? Eigentlich dürfte das nicht der Fall sein; 'zufällig' heißt, daß es kein Schema gibt.

In Wahrheit ist **RND** nicht völlig zufällig, weil es sich an eine feste Folge von 65536 Zahlen hält. Diese sind aber so gründlich durcheinandergeschüttelt, daß es zumindest keine auffällige Schematik gibt; wir sagen, **RND** sei *pseudo-zufällig.*

RND liefert eine Zufallszahl zwischen Ø und 1, aber Sie können leicht Zufallszahlen in anderen Bereichen erhalten. Beispiel: **5*RND** liegt zwischen Ø und 5, **1.3+0.7*RND** zwischen 1.3 und 2. Verwenden Sie, um ganze Zahlen zu erhalten, **INT** (ohne zu vergessen, daß **INT** stets abrundet) wie bei **1+INT (RND*6)**, das wir in einem Programm verwenden, mit dem das Würfeln simuliert werden soll. **RND*6** liegt im Bereich Ø bis 6, aber da 6 nie erreicht wird, ist **INT (RND*6)** Ø, 1, 2, 3, 4 oder 5.

Hier das Programm:

```
1Ø REM Würfelprogramm
2Ø CLS
3Ø FOR n=1 TO 2
4Ø PRINT 1+INT (RND*6);" ";
5Ø NEXT n
6Ø INPUT a$: GO TO 2Ø
```

Drücken Sie jedesmal, wenn Sie würfeln wollen, **ENTER.**

Die **RANDOMIZE**-Anweisung wird dazu verwendet, um **RND** bei seiner Zahlenfolge an einem bestimmten Punkt beginnen zu lassen, wie Sie aus diesem Programm ersehen:

```
10 RANDOMIZE 1
20 FOR n=1 TO 5: PRINT RND ,: NEXT n
30 PRINT: GO TO 10
```

Nach jeder Ausführung von **RANDOMIZE 1** fängt die **RND**-Folge erneut mit 0.0022735596 an. Sie können in der **RANDOMIZE**-Anweisung andere Zahlen zwischen 1 und 65535 verwenden, um die **RND**-Folge an verschiedenen Stellen beginnen zu lassen.

Wenn Sie ein Programm hätten, das **RND** enthält, und sich darin Fehler befänden, die Sie nicht gefunden haben, wäre es nützlich, **RANDOMIZE** so zu verwenden, damit das Programm sich bei jedem Lauf gleich verhält.

RANDOMIZE allein (**RANDOMIZE 0** hat dieselbe Wirkung) ist anders, weil es **RND** wirklich zufällig macht. Das sehen Sie beim nächsten Programm:

```
10 RANDOMIZE
20 PRINT RND: GO TO 10
```

Die Folge, die Sie hier haben, ist nicht sehr zufällig, weil **RANDOMIZE** die Zeit benützt, seitdem der Computer eingeschaltet ist. Da das bei jeder Ausführung von **RANDOMIZE** um denselben Betrag erhöht worden ist, bewirkt das nächste **RND** mehr oder weniger dasselbe. Eine bessere Zufallsauswahl erhalten Sie, wenn Sie **GO TO 10** durch **GO TO 20** ersetzen.

Zur Beachtung: Die meisten Abarten von BASIC verwenden **RND** und **RANDOMIZE**, um Zufallszahlen zu erzeugen, aber sie verwenden sie nicht alle auf dieselbe Weise.

Hier ein Programm, um Münzen zu werfen und zu zählen, wie oft Kopf oder Zahl herauskommt.

```
10 LET Kopf=0: LET Zahl=0
20 LET Münze=INT (RND*2)
30 IF Münze=0 THEN LET Kopf=Kopf+1
40 IF Münze=1 THEN LET Zahl=Zahl+1
50 PRINT Kopf;";";Zahl,
60 IF ZAHL <>0 THEN PRINT Kopf/Zahl;
70 PRINT: GO TO 20
```

Das Verhältnis Kopf zu Zahl sollte annähernd bei 1 liegen, wenn Sie lange genug weitermachen, weil man auf lange Sicht annähernd gleiches Vorkommen von Kopf und Zahl erwartet.

Übungen

1. Testen Sie diese Regel:
   Angenommen, Sie wählen eine Zahl zwischen 1 und 872 und schreiben

   RANDOMIZE Ihre Zahl

   Dann wird der nächste Wert von **RND** betragen

(75*(Ihre Zahl+1)−1)/65536

2. (Nur für Mathematiker)
   Es soll p eine (große) Primzahl sein und a eine primitive Wurzel modulo p. Wenn b_i der Rest von a^i modulo p ($1 \leq b_i \leq p-1$), ist die Folge

$$\frac{b_i - 1}{p-1}$$

eine zyklische Folge von $p-1$ ungleichen Zahlen im Bereich 0 bis 1 (außer 1). Wird a in geeigneter Weise ausgewählt, können diese Zahlen als recht zufällig erscheinen.

65537 ist eine Fermatsche Primzahl, $2^{16}+1$. Da die multiplikative Gruppe von nicht Null betragenden Resten modulo 65537 eine Potenz von 2 als ihre Ordnung hat, ist ein Rest eine primitive Wurzel dann und nur dann, wenn er kein quadratischer Rest ist. Verwenden Sie das Gaußsche Gesetz der quadratischen Reziprozität, um zu zeigen, daß 75 eine primitive Wurzel modulo 65537 ist.

Der ZX Spectrum verwendet $p=65537$ und $a=75$ und speichert b_i-1. **RND** umfaßt die Ersetzung von b_i-1 im Speicher durch $b_{i+1}-1$ und liefert das Resultat $(b_{i+1}-1),(p-1)$.
RANDOMIZE n (mit $1 \leq n \leq 65535$) ergibt b_i gleich $n+1$.
RND ist über den Bereich 0 bis 1 annähernd gleich verteilt.

KAPITEL 12

Arrays (Variablenfelder)

Zusammenfassung:
Arrays (die Art, wie der ZX Spectrum Stringarrays behandelt, entspricht nicht ganz der Norm).
DIM ...

Angenommen, Sie haben eine Liste von Zahlen, etwa die Noten von zehn Schülern einer Klasse. Um sie im Computer zu speichern, könnten Sie für jede Person eine eigene Variable aufstellen, aber das wäre sehr umständlich. Sie könnten die Variable Knacke 1, Knacke 2 und so weiter bis zu Knacke 10 nennen, aber das Programm, das diese zehn Zahlen aufstellen würde, wäre ziemlich lang und mühsam einzugeben.

Wieviel schöner wäre es, wenn Sie so schreiben könnten:

```
 5 REM Dieses Programm läuft nicht
10 FOR n=1 TO 10
20 READ Block n
30 NEXT n
40 DATA 10,2,5,19,16,3,11,1,0,6
```

Das geht aber nicht.

Allerdings gibt es eine Methode, mit deren Hilfe Sie diese Idee in die Tat umsetzen können. Sie verwendet Arrays (Felder, Tabellen). Ein Array ist eine Menge von Variablen, seine Elemente, alle mit demselben Namen, unterschieden nur durch eine Zahl (den *Index*), in Klammern dem Namen angefügt. In unserem Beispiel könnte der Name *b* sein (wie bei Steuervariablen von **FOR – NEXT**-Schleifen muß der Name eines Arrays ein Einzelbuchstabe sein). Die zehn Variablen wären dann *b*(1), *b*(2), und so weiter bis zu *b*(10).

Die Elemente eines Arrays werden *indizierte* Variable genannt, im Gegensatz zu den *einfachen* Variablen, die Sie schon kennen.

Bevor Sie ein Array verwenden können, müssen Sie im Computer Platz dafür reservieren. Das tun Sie mit einer **DIM** (für Dimensioniere)-Anweisung.

```
DIM b(10)
```

stellt ein Array genannt *b* mit der Dimension 10 auf (das heißt, es gibt 10 indizierte Variablen *b*(1),..., *b*(10)), und initialisiert die 10 Werte auf 0. Außerdem löscht es jedes vorher vorhandene Array namens *b*. (Aber nicht eine einfache Variable. Ein Array und eine einfache numerische Variable mit demselben Namen können gleichzeitig vorhanden sein, und es sollte zwischen ihnen keine Verwirrung geben, weil die Arrayvariable stets einen Index hat.)

Der Index kann ein willkürlich benannter numerischer Ausdruck sein, also können Sie jetzt schreiben

```
10 FOR n=1 TO 10
20 READ b(n)
30 NEXT n
40 DATA 10,2,5,19,16,3,11,1,0,6
```

Sie können auch Arrays mit mehr als einer Dimension aufstellen. Bei einem zweidimensionalen Array brauchen Sie zwei Zahlen, um eines der Elemente zu bestimmen – ganz ähnlich wie bei den Zeilen- und Spaltennummern, die auf dem Fernseh-Bildschirm eine Zeichenposition bestimmen – so daß es die Form einer Tabelle hat. Oder Sie könnten sich die Zeilen- und Spaltennummern (zwei Dimensionen) auf eine gedruckte Seite bezogen vorstellen und eine zusätzliche Dimension für die Seitennummer nehmen. Wir sprechen natürlich von numerischen Arrays; die Elemente wären nicht als gedruckte Zeichen in einem Buch, sondern Zahlen. Stellen Sie sich die Elemente eines dreidimensionalen Arrays *v* als bestimmt vor durch *v* (Seitennummer, Zeilennummer, Spaltennummer).

Beispiel: Um ein zweidimensionales Array *c* mit den Dimensionen 3 und 6 aufzustellen, verwenden Sie eine **DIM**-Anweisung

DIM c(3,6)

Dann erhalten Sie 3*6=18 indizierte Variable

c(1,1),c(1,2),. . .,c(1,6)
c(2,1),c(2,2),. . .,c(2,6)
c(3,1),c(3,2),. . .,c(3,6)

Dasselbe Prinzip funktioniert bei jeder beliebigen Zahl von Dimensionen.

Sie können zwar eine Zahl und ein Array mit demselben Namen verwenden, nicht aber zwei Arrays mit demselben Namen, selbst wenn sie verschiedene Dimensionszahlen haben.

Es gibt auch Stringarrays. Die Strings in einem Array unterscheiden sich von einfachen Strings dadurch, daß sie von festgelegter Länge sind und die Zuteilung zu ihnen stets nach dem Prokrustesbett-System erfolgt – abhacken oder mit Leerräumen auffüllen. Man kann sie sich auch vorstellen als Arrays von Einzelzeichen (mit einer zusätzlichen Dimension). Der Name eines Stringarrays ist ein Einzelbuchstabe, gefolgt von **$**, und ein Stringarray und eine einfache Stringvariable können nicht denselben Namen tragen (das ist anders als bei den Zahlen).

Nehmen wir an, Sie möchten ein Array **a$** von fünf Strings. Sie müssen entscheiden, wie lang diese Strings sein sollen. Unterstellen wir, je 1Ø Zeichen wären genug. Dann heißt es

DIM a$(5,1Ø) (geben Sie das ein)

Das stellt ein Array von 5*1Ø Zeichen auf, aber Sie können sich jede Reihe auch als einen String vorstellen:

a$(1)=a$(1,1), a$(1,2),. . .,a$(1,1Ø)
a$(2)=a$(2,1), a$(2,2),. . .,a$(2,1Ø)
. . . . . .
a$(5)=a$(5,1), a$(5,2),. . .,a$(5,1Ø)

Wenn Sie dieselbe Zahl von Indices eingeben (in diesem Fall zwei), wie die **DIM**-

Anweisung an Dimensionen enthält, erhalten Sie ein Einzelzeichen, aber wenn Sie das letzte weglassen, erhalten Sie ein String von festgelegter Länge. So ist beispielsweise A$(2,7) das siebte Zeichen im String a$(2); in der Slicing-Schreibweise könnten wir das auch schreiben als A$(2)(7). Tippen Sie nun

```
LET a$(2)="1234567890"
```

und

```
PRINT a$(2),a$(2,7)
```

Sie erhalten

```
1234567890        7
```

Für den letzten Index (der, den Sie weglassen können) können Sie auch einen Slice-Ausdruck nehmen, so daß zum Beispiel

```
a$(2,4 TO 8)=a$(2)(4 TO 8)="45678"
```

Merke:
In einem Stringarray haben alle Strings dieselbe festgelegte Länge.
Die **DIM**-Anweisung hat eine zusätzliche Zahl (die letzte), um die Länge festzulegen.
Wenn Sie eine indizierte Variable für ein Stringarray schreiben, können Sie eine zusätzliche Zahl oder einen Slice-Ausdruck als Entsprechung zu der zusätzlichen Zahl in der **DIM**-Anweisung einfügen.

Es gibt auch Stringarrays ohne Dimensionen. Nehmen Sie

```
DIM a$(10)
```

wobei Sie feststellen werden, daß **a$** sich genau wie eine Stringvariable verhält, sieht man davon ab, daß es stets die Länge 10 hat und die Zuteilung stets dem Prokrustesprinzip entspricht.

Übungen

1. Verwenden Sie **READ**- und **DATA**-Anweisungen, um ein Array **m$** von zwölf Strings aufzustellen, in dem **m$(n)** der Name des **n**-ten Monats ist. (Ein Tip: Die **DIM**-Anweisung lautet **DIM m$(12,9)**. Testen Sie das durch Anzeige des gesamten **m$(n)** (verwenden Sie eine Schleife)).

Schreiben Sie

```
PRINT "Das ist der schöne Monat ";m$(5);" ,
"der alles uns macht neu"
```

Was läßt sich gegen die vielen Leerräume tun?

KAPITEL 13

Bedingungen

Zusammenfassung:
AND, OR
NOT

In Kapitel 3 haben wir gesehen, daß ein **IF**-Befehl die Form annimmt

IF Bedingung **THEN** ...

Die Bedingungen dort waren die Beziehungen =, <, >, <=, >= und <>, die zwei Zahlen oder zwei Strings vergleichen. Sie können davon auch mehrere kombinieren, wenn Sie die logischen Operationen **AND, OR** und **NOT** verwenden.

Eine Beziehung **AND** eine andere Beziehung sind wahr, sobald beide Beziehungen wahr sind, also können Sie eine Zeile schreiben wie

```
IF a$="ja" AND x>0 THEN PRINT x
```

worauf **x** nur dann angezeigt wird, wenn **a$**="ja" und **x**>0. Das BASIC ist hier dem Englischen so nah, daß es kaum zu lohnen scheint, die Einzelheiten zu erläutern. (Im Deutschen: Wenn a$="ja" und x größer Null, dann zeige x an.) Wie im Englischen (übertragen im Deutschen) können Sie viele Beziehungen mit einem **AND** zusammenfügen, und das Ganze ist dann wahr, wenn alle einzelnen Beziehungen wahr sind.

Eine Beziehung **OR** andere Beziehung ist wahr, sobald mindestens eine der beiden Beziehungen wahr ist. (Merke: Sie ist auch dann wahr, wenn beide Beziehungen wahr sind, was im Englischen und auch im Deutschen nicht immer zutrifft.)

Die **NOT**-Beziehung kehrt die Dinge um. Die **NOT**-Beziehung ist dann wahr, sobald die Beziehung unwahr ist, und unwahr, sobald sie wahr ist!

Mit Beziehungen und **AND**, **OR** und **NOT** kann man logische Ausdrücke ebenso bilden, wie man mit Zahlen und **+, –** und so weiter numerische Ausdrücke zu bilden vermag; wenn das nötig ist, kann man sie sogar in Klammern stellen. Sie haben auf dieselbe Weise Prioritäten wie die üblichen Operationen **+**, **–**, *, / und ↑: **OR** hat die niedrigste Priorität, dann kommt **AND**, dann **NOT**, dann kommen die Beziehungen und die üblichen Operationen.

NOT ist eigentlich eine Funktion mit einem Argument und einem Resultat, aber seine Priorität liegt viel niedriger als bei den anderen Funktionen. Aus diesem Grund benötigt sein Argument keine Klammern, wenn nicht **AND** oder **OR** (oder beide) enthalten sind. **NOT a=b** bedeutet dasselbe wie **NOT (a=b)** (und natürlich dasselbe wie **a<>b**, versteht sich).

<> ist die Verneinung von = in dem Sinn, daß es dann und nur dann wahr ist, wenn = unwahr ist. Mit anderen Worten

a<>b ist dasselbe wie **NOT a=b**

und ferner

NOT a<>b ist dasselbe wie **a=b**

Machen Sie sich klar, daß >= und <= die Verneinungen von < und > sind. Auf diese Weise können Sie **NOT** von einer Beziehung immer loswerden, indem Sie die Beziehung ändern.

Außerdem ist

NOT (ein erster logischer Ausdruck **AND** ein zweiter)

dasselbe wie

NOT (das erste) **OR NOT** (das zweite)

und

NOT (ein erster logischer Ausdruck **OR** ein zweiter)

dasselbe wie

NOT (das erste) **AND NOT** (das zweite)

Damit können Sie **NOT**-Anweisungen durch Klammern verwenden, bis sie schließlich alle auf Beziehungen angewendet werden, und sie dann loswerden. Logisch gesprochen ist **NOT** überflüssig, obwohl Sie trotzdem den Eindruck haben können, daß ein Programm dadurch klarer wird.

Der folgende Abschnitt ist ziemlich kompliziert und darf von den Ängstlichen übersprungen werden!

Schreiben Sie

PRINT 1=2,1<>2

Sie mögen meinen, hier müßte ein Syntaxfehler angezeigt werden. Für den Computer gibt es aber keinen logischen Wert. Statt dessen verwendet er gewöhnliche Zahlen, die einigen Regeln unterworfen sind.

I) =, >, <, <=, >= und <> liefern alle numerische Resultate: 1 für wahr, und Ø für unwahr. So zeigte der obige **PRINT**-Befehl Ø für '1=2' an, das unwahr ist, und 1 für '1<>2', das wahr ist.

II) In

IF Bedingung **THEN** ...

kann die Bedingung jeder numerische Ausdruck sein. Wenn der Wert Ø ist, zählt er als unwahr, jeder andere Wert (eingeschlossen der Wert 1, den eine wahre Beziehung liefert) zählt als wahr. So bedeutet die **IF**-Anweisung genau dasselbe wie

IF Bedingung <>Ø **THEN** ...

III) **AND, OR** und **NOT** sind auch Operationen, die wie Zahlenwerte behandelt werden.

x **AND** y hat den Wert
- *x*, wenn *y* wahr (nicht Null)
- Ø (unwahr), wenn *y* falsch (Null)

x **OR** y hat den Wert
- 1 (wahr), wenn *y* wahr (nicht Null)
- *x*, wenn *y* falsch (Null)

NOT x hat den Wert
- Ø (falsch), wenn *x* wahr (nicht Null)
- 1 (wahr), wenn *x* falsch (Null)

(Beachten Sie, daß 'wahr' dann 'nicht Null' bedeutet, wenn wir einen gegebenen Wert

prüfen, aber '1', wenn wir einen neuen hervorbringen.)

Lesen Sie das Kapitel im Licht dieser Offenbarung noch einmal durch, und vergewissern Sie sich, daß alles klappt.

In den Ausdrücken **x AND y**, **x OR y** und **NOT x** nehmen **x** und **y** in der Regel die Werte Ø und 1 für unwahr und wahr an. Errechnen Sie die zehn verschiedenen Kombinationen (vier für **AND**, vier für **OR** und zwei für **NOT**) und prüfen Sie, daß sie leisten, was Sie nach dem Kapitel erwarten können.

Probieren Sie das folgende Programm aus:

```
10 INPUT a
20 INPUT b
30 PRINT (a AND a>=b)+(b AND a<b)
40 GO TO 10
```

Jedesmal wird die größere der beiden Zahlen **a** und **b** angezeigt.

Überzeugen Sie sich davon, daß Sie sich

x AND y

so vorstellen können, als bedeute es

x wenn **y** (sonst ist das Resultat Ø)

und

x OR y

bedeute

x falls nicht **y** (in diesem Fall ist das Resultat 1)

Ein Ausdruck, der **AND** und **OR** so verwendet, wird *bedingter* Ausdruck genannt. Ein Beispiel mit **OR** könnte lauten

```
LET Gesamtpreis=Preis ohne Steuer*(1.13 OR v$="mit Null bewertet")
```

Beachten Sie, wie **AND** zur Addition neigt (weil sein Fehlerwert Ø ist) und **OR** zur Multiplikation (weil sein Fehlerwert 1 ist).

Sie können auch bedingte Ausdrücke verwenden, die als Strings bewertet werden, aber nur mit **AND.**

x$ AND y hat den Wert

x$, wenn y nicht Null
"", wenn y Null

was also heißt **x$**, wenn **y** (sonst der leere String).

Versuchen Sie es mit diesem Programm, das zwei Strings eingibt und sie in alphabetische Reihenfolge bringt:

prüfen, aber '1', wenn wir einen neuen hervorbringen.)

Lesen Sie das Kapitel im Licht dieser Offenbarung noch einmal durch, und vergewissern Sie sich, daß alles klappt.

In den Ausdrücken **x AND y**, **x OR y** und **NOT x** nehmen **x** und **y** in der Regel die Werte Ø und 1 für unwahr und wahr an. Errechnen Sie die zehn verschiedenen Kombinationen (vier für **AND**, vier für **OR** und zwei für **NOT**) und prüfen Sie, daß sie leisten, was Sie nach dem Kapitel erwarten können.

Probieren Sie das folgende Programm aus:

```
10 INPUT a
20 INPUT b
30 PRINT (a AND a>=b)+(b AND a<b)
40 GO TO 10
```

Jedesmal wird die größere der beiden Zahlen **a** und **b** angezeigt.

Überzeugen Sie sich davon, daß Sie sich

x AND y

so vorstellen können, als bedeute es

x wenn **y** (sonst ist das Resultat Ø)

und

x OR y

bedeute

x falls nicht **y** (in diesem Fall ist das Resultat 1)

Ein Ausdruck, der **AND** und **OR** so verwendet, wird *bedingter* Ausdruck genannt. Ein Beispiel mit **OR** könnte lauten

```
LET Gesamtpreis=Preis ohne Steuer*(1.13 OR v$="mit Null
bewertet")
```

Beachten Sie, wie **AND** zur Addition neigt (weil sein Fehlerwert Ø ist) und **OR** zur Multiplikation (weil sein Fehlerwert 1 ist).

Sie können auch bedingte Ausdrücke verwenden, die als Strings bewertet werden, aber nur mit **AND**.

x$ AND y hat den Wert

x$, wenn *y* nicht Null
"", wenn *y* Null

was also heißt **x$**, wenn **y** (sonst der leere String).

Versuchen Sie es mit diesem Programm, das zwei Strings eingibt und sie in alphabetische Reihenfolge bringt:

KAPITEL 14

Der Zeichenvorrat

Zusammenfassung:
CODE, CHR$
POKE, PEEK
USR
BIN

Die Buchstaben, Ziffern, Interpunktionszeichen und so weiter, die in Strings erscheinen können, werden Zeichen genannt und stellen das Alphabet oder den *Zeichenvorrat* dar, den der ZX Spectrum verwendet. Die meisten dieser Zeichen sind einzelne *Symbole*, aber es gibt noch andere, sogenannte *Token*, die für ganze Wörter stehen, etwa **PRINT, STOP**, <> und so fort.

Es gibt 256 Zeichen. Jedes hat einen Code zwischen 0 und 255. Eine vollständige Liste finden Sie in Anhang A. Um zwischen Codes und Zeichen umzuwandeln, gibt es zwei Funktionen, **CODE** und **CHR$**.

CODE wird auf einen String angewandt und liefert den Code des ersten Zeichens im String (oder 0, wenn der String leer ist).

CHR$ wird auf eine Zahl angewandt und liefert den Einzelzeichenstring, dessen Code diese Zahl ist.

Das nachstehende Programm zeigt den gesamten Zeichenvorrat an.

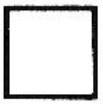
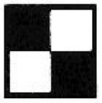
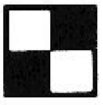
```
10 FOR a=32 TO 255: PRINT CHR$ a;: NEXT a
```

Oben sehen Sie eine Leerstelle, 15 Symbole und Interpunktionszeichen, die zehn Ziffern, sieben weitere Symbole, die Großbuchstaben, sechs Symbole, die Kleinbuchstaben und noch einmal fünf Symbole. Sie alle (außer £ und ©) stammen aus einem vielverwendeten Zeichenvorrat, bekannt unter dem Namen ASCII (für American Standard Codes for Information Interchange = amerikanischer Standardcode für Informationsaustausch). ASCII teilt diesen Zeichen auch numerische Codes zu. Das sind die Codes, die der ZX Spectrum verwendet.

Die restlichen Zeichen sind nicht Teil des ASCII und nur dem ZX Spectrum eigentümlich. An erster Stelle sind das ein Leerraum und 15 Muster von schwarzen und weißen Flecken. Sie werden *Grafiksymbole* genannt und können zum Zeichnen von Bildern verwendet werden. Sie können diese Zeichen über die Tastatur eingeben und dazu den sogenannten *Grafik*modus verwenden. Wenn Sie **GRAPHICS** drücken (**CAPS SHIFT** mit **9**), verändert sich der Cursor zu **G**. Die Tasten für die Ziffern 1 bis 8 liefern dann die Grafiksymbole – allein die Symbole, die auf den Tasten zu sehen sind, zusammen mit einer der beiden Umschalttasten (Shift) dasselbe Symbol, aber in Negativschrift, das heißt, schwarz wird weiß und umgekehrt.

Ohne Rücksicht auf Umschaltungen führt Taste 9 Sie wieder zum normalen (L) Modus zurück, Ziffer 0 ist **DELETE** (löschen).

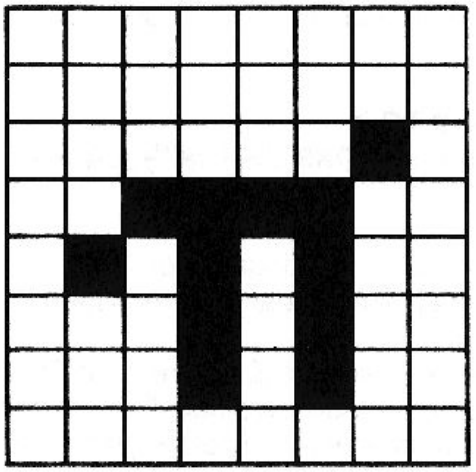
Hier sind die 16 Grafiksymbole:

Symbol	*Code*	*Tasten*	*Symbol*	*Code*	*Tasten*
	128	G 8		143	G umgeschaltet 8
	129	G 1		142	G umgeschaltet 1
	130	G 2		141	G umgeschaltet 2
	131	G 3		140	G umgeschaltet 3
	132	G 4		139	G umgeschaltet 4
	133	G 5		138	G umgeschaltet 5
	134	G 6		137	G umgeschaltet 6
	135	G 7		136	G umgeschaltet 7

Nach den Grafiksymbolen sehen Sie im Zeichenvorrat auf dem Bildschirm offenbar erneut das Alphabet von A bis U. Das sind Zeichen, die Sie selbst wählen können. Wenn der Computer eingeschaltet wird, erscheinen sie aber noch als Buchstaben – man nennt sie *vom Benutzer wählbare Grafik*. Sie können sie über die Tastatur eingeben, indem Sie in den Grafikmodus übergehen und dann die Buchstabentasten A bis U verwenden.

Ein neues Zeichen wählen Sie nach dem folgenden Rezept: Das Beispiel definiert ein Zeichen π.

(I) Arbeiten Sie aus, wie das Zeichen aussieht. Jedes Zeichen besteht aus einem Quadrat von 8 mal 8 Punkten. Jeder Punkt kann entweder die Paper- oder die Inkfarbe annehmen (siehe das Anleitungsheft). Sie zeichnen am besten ein Diagramm der folgenden Art und nehmen schwarze Quadrate für die Inkfarbe:



Wir haben außen einen Rand von 1 Quadrat gelassen, weil die anderen Buchstaben auch alle einen haben (ausgenommen die Kleinbuchstaben mit Unterstrich, wo der Unterstrich bis ganz hinunter geht).
(II) Arbeiten Sie aus, welche vom Benutzer gewählte Grafiktaste π zeigen soll. Nehmen wir die Taste P. Wenn Sie dann im Grafikmodus P drücken, erscheint π.
(III) Speichern Sie das neue Zeichen. Das Muster jedes vom Benutzer gewählte Grafikzeichen wird gespeichert als acht Zahlen, für jede Reihe eine. Sie können jede dieser Zahlen als **BIN**, gefolgt von acht Nullen oder Einsen schreiben – Ø für Paper, 1 für Ink – so daß die acht Zahlen für unser Zeichen so lauten

```
BIN ØØØØØØØØ
BIN ØØØØØØØØ
BIN ØØØØØØ1Ø
BIN ØØ1111ØØ
BIN Ø1Ø1Ø1ØØ
BIN ØØØ1Ø1ØØ
BIN ØØØ1Ø1ØØ
BIN ØØØØØØØØ
```

(Wenn Sie sich mit Binärzahlen auskennen, sollte Ihnen der Hinweis nützlich sein, daß **BIN** dazu verwendet wird, eine Zahl binär, statt, wie üblich, dezimal zu schreiben.)

Diese acht Zahlen werden im Speicher an acht Stellen abgelegt, von denen jede eine *Adresse* hat. Die Adresse des ersten Byte, wie man eine Gruppe von sechs Ziffern nennt, ist **USR "P"** (P deshalb, weil wir es unter II) gewählt haben, die des zweiten **USR "P"+1**, und so weiter bis zur achten mit der Adresse **USR "P"+7**.

USR ist hier eine Funktion, die ein Stringargument in die Adresse des ersten Byte im Speicher für das zugehörige, vom Benutzer gewählte Grafikzeichen verwandelt. Das Stringargument muß ein einzelnes Zeichen sein, entweder das vom Benutzer gewählte Grafikzeichen selbst oder der dazugehörige Buchstabe (groß oder klein geschrieben). Es gibt noch eine andere Verwendung für **USR**, wenn das Argument eine Zahl ist. Damit befassen wir uns noch.

Selbst wenn Sie das nicht begreifen sollten, übernimmt das folgende Programm diese Aufgabe für Sie:

```
10 FOR n=0 TO 7
20 INPUT Reihe: POKE USR "P"+n,Reihe
30 NEXT n
```

Das Programm hält achtmal für die Eingabe von Daten an, damit Sie die obigen acht **BIN**-Zahlen eingeben können. Tun Sie das in der richtigen Reihenfolge, und fangen Sie mit der obersten Reihe an.

Die **POKE**-Anweisung speichert eine Zahl direkt am Speicherplatz und umgeht die von BASIC sonst verwendeten Mechanismen. Das Gegenteil von **POKE** ist **PEEK**. Damit können wir uns den Inhalt eines Speicherplatzes ansehen; der Inhalt dieses Speicherplatzes wird dadurch aber nicht verändert. Mit den beiden Anweisungen befassen wir uns eingehender in Kapitel 24.

Nach den vom Benutzer gewählten Grafikzeichen kommen die Token.

Es wird Ihnen aufgefallen sein, daß wir nicht die ersten 32 Zeichen mit den Codes 0 bis 31 angezeigt bekommen haben. Das sind *Steuer*zeichen. Sie liefern nichts Anzeigbares, haben aber eine weniger greifbare Wirkung auf das Fernsehgerät oder werden dafür verwendet, etwas anderes als den Fernseher zu steuern. Der Fernseher zeigt **?** an, um darauf hinzuweisen, daß er sie nicht versteht. Genauer werden sie in Anhang A beschrieben.

Drei davon, die der Fernseher verwendet, sind die mit den Codes 6, 8 und 13; im Grunde ist **CHR$ 8** das einzige, mit dem Sie etwas werden anfangen können.

CHR$ 6 zeigt Leerstellen genau auf dieselbe Weise an, wie es ein Komma in einer **PRINT**-Anweisung tut. Beispiel:

```
PRINT 1; CHR$ 6; 2
```

leistet dasselbe wie

```
PRINT 1,2
```

Offenkundig kein kluger Weg, sich das nutzbar zu machen. Eine raffinierte Methode ist

```
LET a$="1"+CHR$ 6+"2"
PRINT a$
```

CHR$ 8 ist 'Rücklauf leer'. Es verschiebt die Anzeigeposition um eine Stelle nach hinten. Probieren Sie dazu

```
PRINT "1234"; CHR$ 8;"5"
```

dann wird **1235** angezeigt.

CHR$ 13 ist 'neue Zeile'. Die Anzeigeposition wird zum Beginn der nächsten Zeile gerückt.

Der Fernseher verwendet auch die Steuerzeichen mit den Codes 16 bis 23; sie werden erklärt in den Kapiteln 15 und 16. Alle Steuerzeichen sind aufgeführt in Anhang A.

Wenn wir die Codes statt der Zeichen verwenden, können wir den Begriff der 'alphabetischen Reihenfolge' auf Strings erweitern, die beliebige Zeichen und nicht nur Buchstaben enthalten. Wenn wir, statt uns das übliche Alphabet mit 26 Buchstaben vorzustellen, das erweiterte Alphabet von 256 Zeichen in derselben Reihenfolge wie ihre Codes verwenden, ist das Prinzip genau dasselbe. Beispielsweise stehen diese Strings in ihrer alphabetischen Reihenfolge des ZX Spectrum. (Beachten Sie die eigenartige Erscheinung, daß Kleinbuchstaben nach den großen kommen, 'a' also nach 'Z'. Außerdem spielen die Leerstellen eine Rolle.)

```
CHR$ 3+"ZOOLOGISCHER GARTEN"
CHR$ 8*"AACHENER MÜNSTER"
"   UUUAHH!"
"(Bemerkung in Klammern)"
"1ØØ"
"129.95 einschl. MWST"
"AASVOGEL"
"Aachen"
"Zoo"
"(Einschaltung)"
"aalen"
"aasvogel"
"zoo"
"zoologie"
```

Hier die Regel für die Feststellung, in welcher Reihenfolge Strings erscheinen. Vergleichen Sie zunächst die ersten Zeichen. Wenn sie sich unterscheiden, ist der Code des einen geringer als der des anderen, der String, aus dem er stammt, der frühere (geringere) der beiden Strings. Wenn sie gleich sind, gehen Sie weiter und vergleichen die nächsten Zeichen. Geht bei diesem Vorgang einer der Strings vor dem anderen zu Ende, dann ist das der frühere; im anderen Fall müssen sie gleich sein.

Die Beziehungen =, <, >, <=, >= und <> werden für Strings ebenso verwendet wie für Zahlen. < bedeutet 'kommt vor' und > 'kommt nach'. Demnach sind

```
"AA Mann"<"AASVOGEL"
"AASVOGEL">"AA Mann"
```

beide wahr.

<= und >= beide genauso wie bei Zahlen, so daß

```
"Derselbe String"<="Derselbe String"
```

wahr ist, aber

```
"Derselbe String"<"Derselbe String"
```

falsch.

Probieren Sie das alles mit dem folgenden Programm aus, das zwei Strings eingibt und sie in die richtige Reihenfolge bringt.

```
10 INPUT "Zwei Strings eingeben:",a$,b$
20 IF a$>b$ THEN LET c$=a$: LET a$=b$: LET b$=c$
30 PRINT a$;" ";
40 IF a$<b$ THEN PRINT "<";: GO TO 60
50 PRINT "="
60 PRINT " ";b$
70 GO TO 10
```

Beachen Sie, wie wir **c$** in Zeile 20 einführen müssen, wenn wir **a$** und **b$** tauschen.

```
LET a$=b$: LET b$=a$
```

hätte nicht die gewünschte Wirkung.

Das nächste Programm stellt vom Benützer gewählte Grafikzeichen auf, um Schachfiguren darzustellen:

P für pawn = Bauer
R für rook = Turm
N für knight = Springer
B für bishop = Läufer
K für king = König
Q für queen = Dame

Schachfiguren

```
  5 LET b=BIN 01111100: LET c=BIN
    00111000: LET d=BIN 00010000
 10 FOR n=1 TO 6: READ p$: REM 6 Figuren
 20 FOR f=0 TO 7: REM lies Figur in 8 Bytes
 30 READ a: POKE USR p$+f,a
 40 NEXT f
 50 NEXT n
100 REM Läufer
110 DATA "b",0,d, BIN 00101000, BIN 01000100
120 DATA BIN 01101110,c,b,0
130 REM König
140 DATA "k",0,d,c,d
150 DATA c, BIN 01000100,c,0
160 REM Turm
170 DATA "r",0, BIN 01010100,b,c
180 DATA c,b,b,0
190 REM Dame
200 DATA "q",0, BIN 01010100, BIN 00101000,d
```

```
210 DATA BIN 01101100,b,b,0
220 REM Bauer
230 DATA „p",0,0,d,c
240 DATA c,d,b,0
250 REM Springer
260 DATA „n",0,d,c, BIN 01111000
270 DATA BIN 00011000,c,b,0
```

Beachten Sie, daß statt **BIN 00000000** einfach 0 verwendet werden kann.

Übungen

1. Stellen Sie das obige Programm auf die Anfangsbuchstaben der deutschen Figurennamen um und geben Sie die entsprechenden **BIN**-Zahlen ein.

2. Stellen Sie sich den Raum für ein Symbol aufgeteilt in vier Viertel vor. Wenn dann jedes Teilquadrat entweder weiß oder schwarz sein kann, gibt es 2×2×2×2=16 Möglichkeiten. Suchen Sie sie alle im Zeichenvorrat.

3. Fahren Sie dieses Programm:

```
10 INPUT a
20 PRINT CHR$ a;
30 GO TO 10
```

Wenn Sie damit experimentieren, werden Sie feststellen, daß **CHR$ a** zur nächsten ganzen Zahl *gerundet* wird; wenn **a** nicht im Bereich 0 bis 255 ist, stoppt das Programm mit der Fehlermeldung **B integer out of range.**

4. Was von beiden ist geringer?

```
"EMIL"
"emil"
```

5. Finden Sie heraus, wie man das Programm für die vom Benutzer wählbaren Grafikzeichen so umstellt, daß statt der **INPUT**-Anweisung **READ**- und **DATA**-Anweisungen genutzt werden.

KAPITEL 15

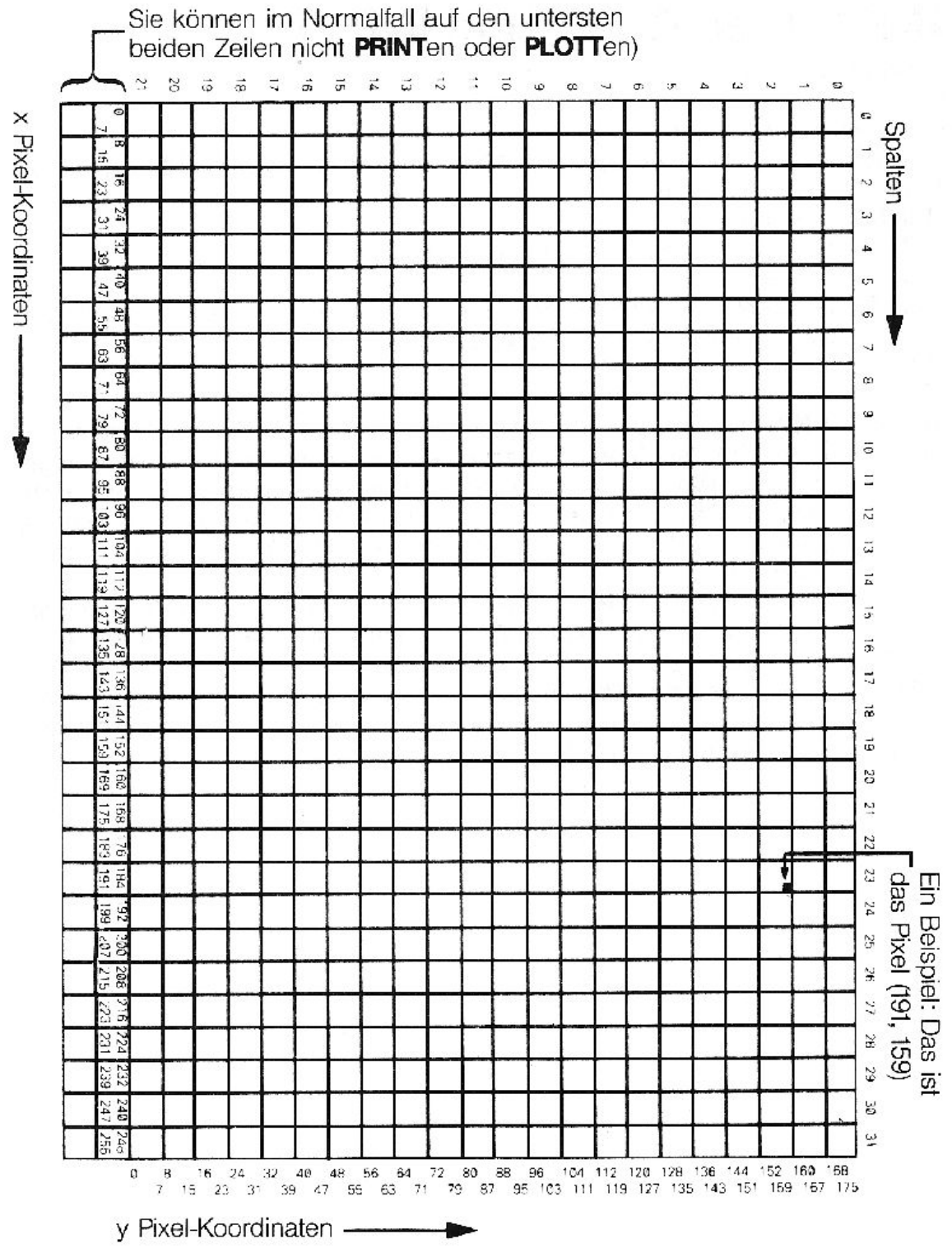
Mehr über PRINT und INPUT

Zusammenfassung:
CLS
PRINT-Posten: gar nichts
Ausdrücke (numerisch oder stringartig): **TAB** numerischer Ausdruck, **AT** numerischer Ausdruck, numerischer Ausdruck
PRINT-Trennsymbole: **, ; '**
INPUT-Posten: Variable (numerisch oder stringartig)
LINE Stringvariable
Jeder Fall von **PRINT,** der nicht mit einem Buchstaben beginnt. (Token werden nicht so aufgefaßt, als fingen sie mit einem Buchstaben an.)
Abrollen.

Sie haben **PRINT** schon vielfach in Verwendung gesehen, werden sich also schon ungefähr ein Bild über den Gebrauch machen können. Ausdrücke, deren Werte angezeigt werden, heißen **PRINT**-*Posten*. Sie werden getrennt durch Kommas oder Strich-Punkte, die man als *Trennsymbole* bezeichnet. Ein **PRINT**-Posten kann auch aus gar nichts bestehen. Damit läßt sich erklären, was passiert, wenn Sie zwei Kommas hintereinander verwenden.

Es gibt noch zwei weitere **PRINT**-Posten. Mit ihnen weist man den Computer an, nicht was, sondern *wo* er anzeigen soll. Beispiel: **PRINT AT 11,16;"*"** stellt mitten auf den Bildschirm einen Stern.

AT Zeile, Spalte versetzt die **PRINT**-Position (die Stelle, wo der nächste Posten angezeigt wird) an die angegebene Zeile und Spalte. Die Zeilen sind numeriert von 0 (oben) bis 21, die Spalten von 0 (links) bis 31.


Sie können im Normalfall auf den untersten beiden Zeilen nicht **PRINT**en oder **PLOTT**en)
x Pixel-Koordinaten
Spalten
Ein Beispiel: Das ist das Pixel (191, 159)
y Pixel-Koordinaten

TAB Spalte stellt Leerstellen her, um die **PRINT**-Position zu der angegebenen Spalte zu führen. Sie bleibt auf derselben Zeile oder geht, wenn sonst zurückgestellt werden müßte, zur nächsten. Merke: Der Computer verringert die Spaltennummer 'modulo 32'. Das heißt, er teilt durch 32 und nimmt den Rest; demnach bedeutet **TAB 33** dasselbe wie **TAB 1.**

Beispiel:

```
PRINT TAB 3Ø;1;TAB 12;"Inhalt"; AT 3,1;"KAPITEL";TAB 24;"Seite"
```

Auf diese Weise könnten Sie ein Inhaltsverzeichnis auf Seite 1 eines Buches überschreiben.

Versuchen Sie das Folgende zu fahren:

```
1Ø FOR n=Ø TO 2Ø
2Ø PRINT TAB 8*n;n;
3Ø NEXT n
```

Das zeigt, was damit gemeint ist, daß die **TAB**-Nummern modulo 32 verringert werden.

Damit das Beispiel eleganter wird: Nehmen Sie in Zeile 2Ø statt der 8 eine 6.

Ein paar kleine Hinweise:

(I) Diese neuen Posten werden am besten mit Strichpunkten abgeschlossen, wie wir es oben getan haben. Sie können Kommas verwenden (oder auch am Ende der Anweisung gar nichts); das bedeutet aber, daß Sie, nachdem Sie die **PRINT**-Position sorgfältig festgelegt haben, diese sofort wieder verschieben, und das ist nicht gerade das Gelbe vom Ei.

(II) Auf den untersten beiden Zeilen (22 und 23) des Bildschirms können Sie nicht anzeigen, weil sie reserviert sind für Befehle, **INPUT**-Daten, Meldungen und so weiter. Wenn auf 'die unterste Zeile' verwiesen wird, ist in der Regel Zeile 21 gemeint.

(III) Mit **AT** können Sie die **PRINT**-Position sogar dorthin stellen, wo schon etwas angezeigt ist; das Alte wird dann gelöscht.

Eine weitere Anweisung im Zusammenhang mit **PRINT** ist **CLS**. Das macht den ganzen Bildschirm leer, dasselbe wird auch mit **CLEAR** und **RUN** bewirkt.

Wenn die Anzeige auf dem Bildschirm unten ankommt, rollt (scrolls) das Ganze nach oben ab, ganz ähnlich wie bei einer Schreibmaschine. Das können Sie sich ansehen, wenn Sie ein paarmal

```
CLS: FOR n=1 TO 22: PRINT n: NEXT n
```

und

```
PRINT 99
```

eingeben.

Wenn der Computer haufenweise Material liefert, achtet er genau darauf, oben nichts wegzurollen, bis sie Gelegenheit gehabt haben, sich das richtig anzusehen. Das können Sie erleben, wenn Sie eingeben:

CLS: FOR n=1 TO 1ØØ: PRINT n: NEXT n

Wenn er den Bildschirm vollgefüllt hat, hört er auf und schreibt unten an den Bildschirm **scroll?**, stellt also die Frage, ob er abrollen soll. Sie können sich jetzt die ersten 22 Zeilen in aller Ruhe ansehen. Wenn Sie damit fertig sind, drücken Sie **y** (für 'yes' = 'ja'), und der Computer liefert den nächsten vollgeschriebenen Bildschirm. Übrigens führt der Druck auf jede beliebige Taste außer **n** (für 'no' = 'nein'), **STOP** (**SYMBOL SHIFT** und **a**) oder **SPACE** (die **BREAK**-Taste) dazu, daß der Computer fortfährt. Die eben genannten Tasten bewirken, daß der Computer das Programm mit der Meldung **D Break – CONT repeats** unterbricht.

Die **INPUT**-Anweisung kann viel mehr, als wir Ihnen bisher verraten haben. Sie kennen bereits **INPUT**-Anweisungen in der Form

INPUT "Wie alt sind Sie?", Alter

wo der Computer die Frage **Wie alt sind Sie?** unten am Bildschirm anzeigt, worauf Sie Ihr Alter eingeben müssen.

Eine **INPUT**-Anweisung besteht in Wirklichkeit genauso aus Posten und Trennsymbolen wie eine **PRINT**-Anweisung. Demzufolge sind **Wie alt sind Sie?** und **Alter** alle beide **INPUT**-Posten. **INPUT**-Posten sind im Grunde dasselbe wie **PRINT**-Posten, aber es gibt ein paar sehr wichtige Unterschiede.

Erstens: Ein offenkundig zusätzlicher **INPUT**-Posten ist die Variable, deren Wert Sie eingeben sollen – in unserem obigen Beispiel also **Alter**. Die Regel: Wenn ein **INPUT**-Posten mit einem Buchstaben anfängt, muß er eine Variable sein, deren Wert einzugeben ist.

Zweitens: Das scheint zu bedeuten, daß Sie die Werte von Variablen nicht als Teil einer Überschrift anzeigen können; das läßt sich aber dadurch umgehen, daß man die Variable in Klammern stellt. Jeder Ausdruck, der mit einem Buchstaben anfängt, muß dann in Klammern stehen, wenn er als Teil einer Überschrift angezeigt werden soll.

Alle Arten von **PRINT**-Posten, die von diesen Regeln nicht betroffen werden, sind gleichzeitig **INPUT**-Posten.

Hier ein Beispiel, um zu veranschaulichen, was vorgeht:

LET mein Alter = INT (RND * 1ØØ): INPUT ("Ich bin ";mein Alter; "."); "Wie alt sind Sie?", Ihr Alter

mein Alter steht in Klammern, also wird der Wert dafür angezeigt. **Ihr Alter** steht nicht in Klammern, also müssen Sie den Wert dafür eingeben.

Alles, was eine **INPUT**-Anweisung anzeigt, geht unten an den Bildschirm, der sich ein wenig unabhängig von der oberen Hälfte aufführt. Vor allem sind seine Zeilen im Verhältnis zur obersten Zeile der unteren Hälfte numeriert, selbst wenn der Fernseh-Bildschirm dadurch in die Höhe geführt wird, was dann eintritt, wenn Sie jede Menge **INPUT**-Daten eingeben.

Wenn Sie sehen wollen, wie **AT** in **INPUT**-Anweisungen wirkt, probieren Sie das:

```
10 INPUT "Das ist Zeile 1.",a$; AT 0,0;"Das ist Zeile 0.",a$; AT 2,0;
   "Das ist Zeile 2.",a$; AT 1,0;"Das ist noch immer Zeile 1.",a$
```

(Bei jedem Anhalten müssen Sie eben **ENTER** drücken.) Wenn **Das ist Zeile 2.** erscheint, rückt der untere Teil des Bildschirms nach oben, um dafür Platz zu machen, aber die Numerierung geht ebenfalls hinauf, so daß die Textzeilen ihre Nummern behalten.

Jetzt probieren Sie:

```
10 FOR n=0 TO 19: PRINT AT n,0;n;: NEXT n
20 INPUT AT 0,0;a$; AT 1,0;a$; AT 2,0;a$; AT 3,0;a$; AT 4,0;a$;
   AT 5,0;a$;
```

Während der untere Teil des Bildschirms ständig nach oben geht, bleibt der obere Teil davon unberührt, bis der untere Teil auf derselben Zeile zu schreiben droht, wo sich die **PRINT**-Position befindet. Dann rollt auch der obere Teil ab, um das zu vermeiden.

Eine weitere Verfeinerung der **INPUT**-Anweisung, die wir auch noch nicht gesehen haben, heißt **LINE**-Input und ist eine andere Methode, Stringvariablen einzugeben. Wenn Sie vor dem Namen einer einzugebenden Stringvariablen **LINE** schreiben, wie bei

```
INPUT LINE a$
```

gibt der Computer Ihnen nicht die String-Anführungszeichen wie sonst bei einer Stringvariablen, obwohl er, was ihn selber betrifft, so tut, als wären sie vorhanden. Wenn Sie also als **INPUT** eingeben

```
Rad
```

erhält **a$** den Wert **Rad.** Da die String-Anführungszeichen am String nicht erscheinen, können Sie sie auch nicht löschen und als **INPUT**-Daten eine andere Art von Stringausdruck eingeben. Denken Sie daran, daß Sie **LINE** nicht für numerische Ausdrücke verwenden können.

Die Steuerzeichen **CHR$ 22** und **CHR$ 23** haben eine ähnliche Wirkung wie **AT** und **TAB**. Als Steuerzeichen sind sie ziemlich merkwürdig: Wenn man eines davon zum Fernseher schickt, damit es angezeigt wird, müssen ihm zwei weitere Zeichen folgen, die nicht ihre übliche Wirkung haben. Sie werden als Zahlen (deren Codes) behandelt, die bei **AT** Zeile und Spalte und bei **TAB** die Tab-Position angeben. Es wird Ihnen leichter fallen, **AT** und **TAB** auf die übliche Weise zu verwenden, statt die Steuerzeichen, aber unter bestimmten Umständen könnten sie nützlich sein. Das Steuerzeichen für **AT** ist **CHR$ 22**. Das erste Zeichen danach bestimmt die Zeilennummer, das zweite die Spaltennummer. Somit hat

```
PRINT CHR$ 22+CHR$ 1+CHR$ c;
```

dieselbe Wirkung wie

```
PRINT AT 1,c;
```

Das ist sogar dann der Fall, wenn **CHR$ 1** oder **CHR$ 2** normalerweise eine andere Bedeutung hätte (beispielsweise, wenn **c=13**); das **CHR$ 22** vorher hebt das auf.

Das Steuerzeichen für **TAB** ist **CHR$ 23**, und die beiden Zeichen danach dienen dazu, eine Zahl zwischen Ø und 65535 festzulegen. Damit wird die Zahl bestimmt, die Sie in einem **TAB**-Posten hätten:

```
PRINT CHR$ 23+CHR$ a+CHR$ b;
```

hat dieselbe Wirkung wie

```
PRINT TAB a+256*b;
```

Mit **POKE** können Sie den Computer daran hindern, daß er die Frage **scroll?** stellt, wenn Sie in Abständen

```
POKE 23692,255
```

eingeben. Anschließend rollt er 255mal ab, bevor er mit **scroll?** unterbricht. Beispiel: Mit

```
1Ø FOR n=Ø TO 1ØØØØ
2Ø PRINT n: POKE 23692,255
3Ø NEXT n
```

können Sie zusehen, wie die Zahlen über den Bildschirm sausen!

Übungen

1. Probieren Sie das folgende Programm bei Kindern aus, um die Kenntnisse im kleinen Einmaleins zu prüfen.

```
1Ø LET m$=" "
2Ø LET a=INT (RND*12)+1: LET b=INT (RND*12)+1
3Ø INPUT (m$); "Wieviel ist ";(a); " * "; (b); " ? ";c
1ØØ IF c=a*b THEN LET m$="Richtig.": GO TO 2Ø
11Ø LET m$="Falsch. Noch einmal.": GO TO 3Ø
```

Wenn sie eine gute Auffassungsgabe besitzen, kommen die Kleinen vielleicht dahinter, daß sie die Berechnungen nicht selbst zu machen brauchen. Beispiel: Wenn der Computer sie nach der Lösung von 2*3 fragt, brauchen sie nur **2*3** einzutippen.

Das kann man umgehen, wenn sie statt Zahlen Strings eingeben müssen. Ersetzen Sie **c** in Zeile 3Ø durch **c$** und in Zeile 1ØØ durch **VAL c$** und fügen Sie die Zeile ein

```
4Ø IF c$ <> STR$ VAL c$ THEN LET m$="Richtig eingeben, als
   Zahl.": GO TO 3Ø
```

Davon lassen sie sich täuschen. Nach ein paar Tagen könnte aber eines der Kinder dahinterkommen, daß sich das umgehen läßt, wenn man die Stringanführungszeichen löscht und eingibt **STR$ (2*3)**. Falls Sie diesen Ausweg auch noch versperren möchten, können Sie **c$** in Zeile 30 durch **LINE c$** ersetzen.

KAPITEL 16

Farben

Zusammenfassung:
INK, PAPER, FLASH, BRIGHT, INVERSE, OVER
BORDER

Fahren Sie dieses Programm:

```
10 FOR m=0 TO 1: BRIGHT m
20 FOR n=1 TO 10
30 FOR c=0 TO 7
40 PAPER c: PRINT "    ";: REM 4 bunte Leerstellen
50 NEXT c: NEXT n: NEXT m
60 FOR m=0 TO 1: BRIGHT m: PAPER 7
70 FOR c=0 TO 3
80 INK c: PRINT c;"    ";
90 NEXT c: PAPER 0
100 FOR c=4 TO 7
110 INK c: PRINT c;"    ";
120 NEXT c: NEXT m
130 PAPER 7: INK 0: BRIGHT 0
```

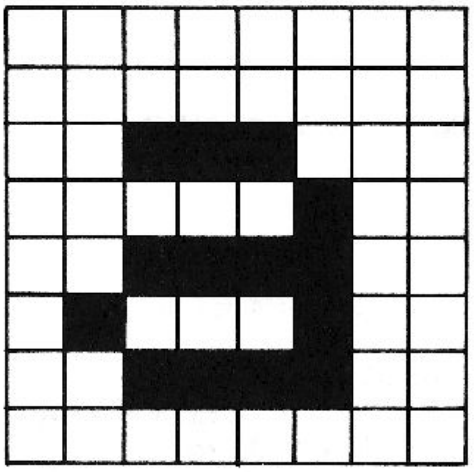
Das zeigt die acht Farben (Weiß und Schwarz eingeschlossen) und die beiden Helligkeitsstufen, die der ZX Spectrum in einem Farbfernseher erzeugen kann. (Beim Schwarzweiß-Gerät sieht man nur verschiedene Grautöne.) Hier eine Liste aller Farben zum Vergleich; über den entsprechenden Zifferntasten stehen sie auch noch einmal.

0 black = schwarz
1 blue = blau
2 red = rot
3 magenta = violett
4 green = grün
5 pale blue (cyan) = hellblau
6 yellow = gelb
7 white = weiß

Bei einem Schwarzweiß-Fernseher stellen die Ziffern die Helligkeitsstufen dar.

Wenn Sie die Farben richtig nutzen wollen, sollten Sie ein bißchen mehr davon verstehen, wie sich das Bild zusammensetzt.

Das Bild ist aufgeteilt in 768 (24 Zeilen zu je 32) Positionen, wo Zeichen angezeigt werden können



und jedes Zeichen als Quadrat von 8 mal 8 Punkten angezeigt wird, wie oben das **a**. Das müßte Sie eigentlich an die vom Benutzer gewählten Grafikzeichen in Kapitel 14 erinnern, wo wir für die weißen Punkte Nullen und für die schwarzen Einsen hatten.

Die Zeichenposition setzt sich auch mit zwei Farben in Verbindung: *Ink*, die Vordergrundfarbe, also die für die schwarzen Punkte in unserem Quadrat, und *Paper*, die Hintergrundfarbe für die weißen. Beim Beginn hat jede Position Ink schwarz und Paper weiß, so daß Schrift schwarz auf weiß erscheint.

Die Zeichenposition hat auch einen Helligkeitswert (normal oder besonders hell) und kann veranlaßt werden, zu blinken oder nicht zu blinken – das Blinken wird dadurch bewirkt, daß man Ink- und Paperfarbe vertauscht. Das Ganze kann in Zahlen umgesetzt werden, so daß eine Zeichenposition dann aufweist

(I) ein Quadrat aus 8 mal 8 Nullen und Einsen, um die Form des Zeichens zu bestimmen, mit 0 für Paper und 1 für Ink,
(II) Ink- und Paperfarben, von denen jede als eine Zahl zwischen 0 und 7 codiert ist,
(III) eine Helligkeit – 0 für normal, 1 für besonders hell, und
(IV) eine Blinkzahl – 0 für normal, 1 für Blinken.

Merke: Da die Ink- und Paperfarben eine ganze Zeichenposition besetzen, können Sie in einem Block von 64 Punkten keinesfalls mehr als zwei Farben haben. Dasselbe gilt für die Helligkeits- und Blinkzahl, die sich auf die gesamte Zeichenposition und nicht auf Einzelpunkte beziehen. Farben, Helligkeit und Blinkzahl an einer Position werden *Attribute* genannt.

Wenn Sie auf dem Bildschirm etwas anzeigen, verändern Sie das Punktmuster an dieser Position; es ist weniger naheliegend, aber doch zutreffend, daß Sie auch die Attribute an dieser Position verändern. Anfangs fällt Ihnen das nicht auf, weil alles mit Ink schwarz auf Paper weiß geschrieben wird (bei normaler Helligkeit, ohne Blinken), aber das können Sie mit den Befehlen **INK, PAPER, BRIGHT** und **FLASH** ändern. Probieren Sie

```
PAPER 5
```

und schreiben Sie dann irgend etwas. Das erscheint auf hellblauem Hintergrund, weil die Paperfarben an den Stellen, wo die Zeichen erscheinen, auf Hellblau (mit der Nummer 5) gesetzt werden.

Bei den anderen geht es genauso, also wird nach

PAPER Zahl zwischen Ø und 7
INK Zahl zwischen Ø und 7
BRIGHT Ø oder 1 } Denken Sie sich Ø als aus,
oder 1 als ein
FLASH Ø oder 1 }

jede Anzeige an allen Zeichenpositionen, die sie benutzt, das entsprechende Attribut setzen. Probieren Sie ein paar Möglichkeiten aus. Sie sollten jetzt erkennen können, wie das Programm am Kapitelanfang funktioniert (denken Sie daran, daß ein Leerraum ein Zeichen ist, bei dem **INK** und **PAPER** dieselbe Farbe haben).

In diesen Anweisungen können Sie noch andere Zahlen verwenden, die weniger direkte Wirkungen haben.

8 kann bei allen vier Anweisungen verwendet werden und bedeutet ‚durchsichtig' in dem Sinn, daß das alte Attribut durchscheint. Nehmen wir an, Sie schreiben

PAPER 8

Keine Zeichenposition wird mit ihrer Paperfarbe je auf 8 gesetzt sein, weil es eine solche Farbe gar nicht gibt. Stattdessen wird die Paperfarbe so belassen, wie sie war, wenn darauf angezeigt wird. **INK 8, BRIGHT 8** und **FLASH 8** wirken bei den anderen Attributen genauso.

9 kann nur bei **PAPER** und **INK** verwendet werden und bedeutet 'Kontrast'. Die Farbe (Ink oder Paper), die Sie dabei verwenden, wird dadurch in Kontrast mit der anderen gebracht, daß sie weiß erscheint, wenn die andere Farbe dunkel ist (schwarz, blau, rot oder violett), und schwarz, wenn die andere Farbe hell ist (grün, hellblau, gelb oder weiß).

Probieren Sie das aus mit

INK 9: FOR c=Ø TO 7: PAPER c: PRINT c: NEXT c

Eindrucksvoller stellen sich die Fähigkeiten dar, wenn man das Programm für farbige Streifen am Anfang fährt und dann eingibt

INK 9: PAPER 8: PRINT AT Ø,Ø;:
FOR n=1 TO 1ØØØ: PRINT n;: NEXT n

Die Inkfarbe wird hier an jeder Position in Kontrast zur alten Paperfarbe gesetzt.

Das Farbfernsehen nutzt die eigentlich sonderbare Tatsache aus, daß das menschliche Auge in Wahrheit nur drei Farben sehen kann, die Primärfarben blau, rot und grün. Die anderen Farben sind Mischungen davon. Beispielsweise entsteht Violett durch Mischen von Blau mit Rot – aus diesem Grund ist der Code dieser Farbe, nämlich 3, die Summe der Codes für Blau und Rot.

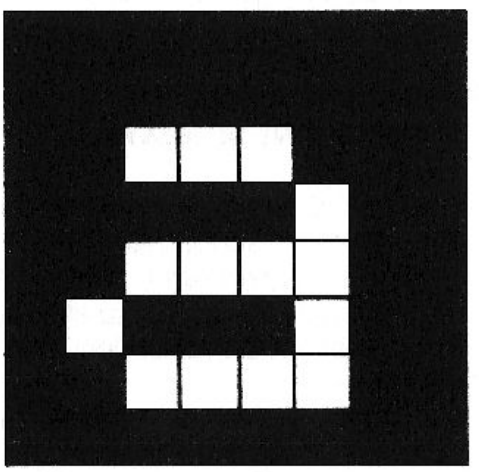
Um ein Bild davon zu bekommen, wie alle acht Farben zusammenpassen, stellen Sie sich drei rechteckige Scheinwerfer in den Farben Blau, Rot und Grün vor, die im

Dunkeln auf ein Blatt weißes Papier leuchten, aber nicht alle genau auf dieselbe Stelle. Wo sie sich überschneiden, sehen Sie Farbmischungen, wie das folgende Programm sie zeigt (Merke: Ink-Leerräume erhält man durch eine der **SHIFT**-Tasten zusammen mit **8** im G-Modus):

```
10 BORDER 0: PAPER 0: INK 7: CLS
20 FOR a=1 TO 6
30 PRINT TAB 6; INK 1;"        ": REM 18
   Inkquadrate
40 NEXT a
50 LET Datenzeile=200
60 GO SUB 1000
70 LET Datenzeile=210
80 GO SUB 1000
90 STOP
200 DATA 2,3,7,5,4
210 DATA 2,2,6,4,4
1000 FOR a=1 TO 6
1010 RESTORE Datenzeile
1020 FOR b=1 TO 5
1030 READ c: PRINT INK c:"      ";: REM 6 Inkquadrate
1040 NEXT b: PRINT : NEXT a
1050 RETURN
```

Es gibt eine Funktion **ATTR**, mit der man feststellen kann, welche Attribute an einer bestimmten Position auf dem Bildschirm vorhanden sind. Da sie einigermaßen kompliziert ist, haben wir sie an den Schluß des Bandes gestellt.

Zwei weitere Anweisungen, **INVERSE** und **OVER**, steuern nicht die Attribute, sondern das Punktemuster, das auf dem Bildschirm angezeigt wird. Sie verwenden die Zahlen 0 für "aus" und 1 für "ein" wie **FLASH** und **BRIGHT**, aber das sind die einzigen Möglichkeiten. Wenn Sie **INVERSE 1** eingeben, erscheinen die angezeigten Punktemuster in der Umkehrung zu vorher: Paperpunkte werden ersetzt durch Inkpunkte und umgekehrt. So würde **a** erscheinen als



Wenn wir (wie beim Einschalten) Ink schwarz und Paper weiß haben, erscheint dieses **a** weiß auf schwarz – aber an dieser Zeichenposition haben wir nach wie vor Ink schwarz und Papier weiß. Verändert haben sich die Punkte.

Die Anweisung

OVER 1

löst eine besondere Art des Überschreibens aus. Normalerweise wird, wenn etwas auf eine Zeichenposition geschrieben wird, das Vorherige völlig gelöscht, aber nun wird das neue Zeichen einfach auf das alte gesetzt (vgl. Übung 1). Das kann besonders nützlich sein, wenn man zusammengesetzte Zeichen schreiben möchte, etwa Buchstaben mit Akzentzeichen, wie im folgenden Programm für deutsche Umlaute. (Drücken Sie vorher **NEW**.)

```
10 OVER 1
20 FOR N=1 TO 32
30 PRINT "o"; CHR$ 8;" "" ";
40 NEXT n
```

(Sie sehen das Steuerzeichen **CHR$ 8,** das um eine Stelle zurückgeht.)

INK, PAPER und so weiter sind noch auf eine andere Weise zu verwenden, die Sie vermutlich nützlicher finden werden als die Anweisungen damit. Sie können sie als Posten in eine **PRINT**-Anweisung stellen (gefolgt von **;**); sie leisten dann genau dasselbe wie als eigenständige Anweisungen, wobei ihre Wirkung nur vorübergehend anhält; das geht bis zum Ende der **PRINT**-Anweisung, die sie enthält. Falls Sie also eingeben

PRINT PAPER 6;"x": PRINT "y"

wird nur das x auf Gelb stehen.

INK und die anderen als Anweisungen beeinflussen nicht die Farben des unteren Bildschirmteils, wo Befehle und **INPUT**-Daten eingegeben werden. Der untere Bildschirmteil verwendet die Farbe des Randes als seine Papierfarbe und Code 9 als Kontrast für seine Inkfarbe: Es herrscht normale Helligkeit und kein Blinken. Sie können die Randfarbe auf jede der acht normalen Farben (nicht 8 oder 9) umstellen, wenn Sie die Anweisung verwenden

BORDER Farbe

Wenn Sie **INPUT**-Daten eingeben, hält sich das an die Regel, kontrastierende Ink auf Paper von Randfarbe zu verwenden, aber Sie können die Farbe der Überschriften verändern, die der Computer schreibt, wenn Sie in der **INPUT**-Anweisung **INK**- und **PAPER** (und so weiter)-Posten verwenden, genau wie in einer **PRINT**-Anweisung. Ihre Wirkung hält entweder bis zum Ende der Anweisung an oder so lange, bis **INPUT**-Daten eingegeben werden, je nachdem, was früher kommt. Nehmen Sie

INPUT FLASH 1; INK 1; "Wie ist Ihre Nummer?";n

Es gibt noch eine weitere Methode, die Farben mit Steuerzeichen zu verändern – ganz ähnlich wie bei den Steuerzeichen für **AT** und **TAB** in Kapitel 15.

CHR$ 16 entspricht **INK**
CHR$ 17 entspricht **PAPER**
CHR$ 18 entspricht **FLASH**
CHR$ 19 entspricht **BRIGHT**
CHR$ 20 entspricht **INVERSE**
CHR$ 21 entspricht **OVER**

Auf jedes folgt ein Zeichen, das eine Farbe mit ihrem Code angibt. Also hat etwa

PRINT CHR$ 16+CHR$ 9; . . .

dieselbe Wirkung wie

PRINT INK 9; . . .

Im Ganzen gesehen, werden Sie sich die Mühe, diese Steuerzeichen zu verwenden, nicht machen, weil Sie ebenso gut die Farbenposten benützen können. Einen Nutzen haben sie aber doch: Sie können sie in Programme stellen. Das führt dazu, daß verschiedene Teile in verschiedenen Farben aufgelistet werden, damit sie voneinander abgesetzt werden oder wenigstens hübsch aussehen. Sie müssen Sie nach der Zeilennummer eingeben, sonst gehen sie einfach verloren.

Um sie ins Programm zu bekommen, müssen Sie sie über die Tastatur eingeben, meistens im erweiterten Modus mit den Ziffern.

Die Ziffern 0 bis 7 setzen die entsprechende Farbe – Ink, wenn auch **CAPS SHIFT** gedrückt wird, Paper, wenn das nicht der Fall ist. Genauer: Wenn Sie im E-Modus sind und eine Ziffer drücken (z.B. 6 für gelb – auf jeden Fall muß es zwischen 0 und 7 liegen, also nicht 8 oder 9), werden zwei Zeichen eingefügt: Zuerst **CHR$ 17** für **PAPER**, und dann **CHR$ 6** mit der Bedeutung 'setze auf gelb'. Wenn Sie beim Drücken der Ziffer **CAPS SHIFT** mitgedrückt hätten, wäre **CHR$ 16** mit der Bedeutung 'setze Inkfarbe' statt **CHR$ 17** gekommen.

Da das zwei Zeichen sind, können Sie eigenartige Wirkungen erzielen, wenn Sie sie löschen – Sie müssen zweimal **DELETE** drücken, und nach dem erstenmal erhalten Sie oft ein Fragezeichen oder gar noch merkwürdigere Dinge. Keine Sorge – drücken Sie einfach ein zweitesmal **DELETE**.

◀ und ▶ verhalten sich manchmal auch sonderbar, während der Cursor an den Steuerzeichen vorbeigeht.

Nach wie vor in erweitertem Modus

liefert **8 CHR$ 19** und **CHR$ 0** normale Helligkeit
liefert **9 CHR$ 12** und **CHR$ 1** stärkere Helligkeit
CAPS SHIFT mit **8** liefert **CHR$ 18** und **CHR$ 19** kein Blinken
CAPS SHIFT mit **9** liefert **CHR$ 19** und **CHR$ 1** Blinken

Im gewöhnlichen L-Modus gibt es noch zwei mehr:

CAPS SHIFT mit **3** liefert **CHR$ 2Ø** und **CHR$ Ø** normale Zeichen
CAPS SHIFT mit **4** liefert **CHR$ 2Ø** und **CHR$ 1** Zeichen in Negativschrift

Zur Zusammenfassung hier eine vollständige Beschreibung der obersten Tastenreihe:

MODUS	UMSCHALTUNG										
E	SYMBOL	**DEF FN**	**FN**	**LINE**	**OPEN #**	**CLOSE #**	**MOVE**	**ERASE**	**POINT**	**CAT**	**FORMAT**
	CAPS	Ink blau	Ink rot	Ink magenta	Ink grün	Ink hellblau	Ink gelb	Ink weiß	Blink aus	Blink ein	Ink schwarz
	OHNE	Papier blau	Papier rot	Papier magenta	Papier grün	Papier hellblau	Papier gelb	Papier weiß	Normal hell	Besonders hell	Papier schwarz
G	BEIDE									Grafik aus	**DELETE**
	OHNE									Grafik aus	**DELETE**
K,L oder C	CAPS	**EDIT**	CAPS LOCK	TRUE VIDEO	INVERSE VIDEO	⇧	⇦	⇨	⇩	Grafik-Modus	**DELETE**
	SYMBOL	!	@	#	$	%	&	'	(	)	_
	OHNE	1	2	3	4	5	6	7	8	9	Ø

Die Funktion **ATTR** hat die Form

ATTR (Zeile, Spalte)

Ihre beiden Argumente sind die Zeilen- und Spaltennummern, die Sie bei einem **AT**-Posten verwenden würden, ihr Resultat ist eine Zahl, die an der entsprechenden Zeichenposition auf dem Bildschirm die Farben und dergleichen zeigt. Sie können sie in Ausdrücken so unbehindert verwenden wie jede andere Funktion auch.

Die Zahl des Resultats ist die Summe von vier anderen Zahlen, nämlich
128, wenn die Zeichenposition blinkt, Ø, wenn nicht
64, wenn die Zeichenposition hell ist, Ø, wenn normal
8* der Code für die Paperfarbe
der Code für die Inkfarbe

Beispiel: Wenn die Zeichenposition blinkt und von normaler Helligkeit mit gelbem Paper und blauer Ink ist, lauten die vier Zahlen, die wir zusammenaddieren müssen, 128, Ø, 8*6=48 und 1, zusammen 177. Prüfen Sie das nach mit

PRINT AT Ø,Ø; FLASH 1; PAPER 6; INK 1;" "; ATTR (Ø,Ø)

Übungen

1. Probieren Sie aus

PRINT "B"; CHR$ 8; OVER 1;"/";

Wo das **/** durch das B gegangen ist, hat es einen weißen Punkt hinterlassen. So wirkt sich das Überschreiben beim ZX Spectrum aus: Zweimal Paper oder zweimal Ink führt zu Paper, eins von jedem zu Ink. Das hat die interessante Eigenschaft, daß Sie, wenn Sie dasselbe zweimal überschreiben, wieder das erhalten, was Sie vorher hatten. Warum erhalten Sie, wenn Sie eingeben

PRINT CHR$ 8; OVER 1;"/"

wieder ein unbeschädigtes **B** zurück?

2. Schreiben Sie

PAPER Ø: INK Ø

Ist es nicht gut, daß sie sich auf den unteren Bildschirmteil nicht auswirken?

Geben Sie dazu jetzt

BORDER Ø

und sehen Sie sich an, wie gut der Computer sich um Sie kümmert!

3. Fahren Sie dieses Programm

```
10 POKE 22527+RND*704, RND*127
20 GO TO 10
```

Lassen wir beiseite, wie das funktioniert; es verändert die Farben der Quadrate auf dem Fernsehschirm, und die **RND** sollen dafür sorgen, daß das zufällig geschieht. Die Diagonalstreifen, die Sie schließlich sehen, sind eine Erscheinung des versteckten Schemas in **RND** – eben jenes Schemas, wodurch nur eine Pseudozufälligkeit erreicht wird.

4. Schreiben oder laden Sie das Schachfiguren-Programm aus Kapitel 14 und geben Sie dann das folgende Programm ein. Es zeichnet ein Diagramm einer Schachstellung mit ihnen.

```
  5 REM zeichne leeres Brett
 10 LET bb=1: LET bw=2: REM rot und blau für Brett
 15 PAPER bw: INK bb: CLS
 20 PLOT 79,128: REM Rand
 30 DRAW 65,0: DRAW 0,-65
 40 DRAW -65,0: DRAW 0,65
 50 PAPER bb
 60 REM Brett
 70 FOR n=0 TO 3: FOR m=0 TO 3
 80 PRINT AT 6+2*n, 11+2*m; "  "
 90 PRINT AT 7+2*n, 10+2m; "  "
100 NEXT m: NEXT n
110 PAPER 8
120 LET pw=6: LET pb=5: REM Farbe der weißen und schwarzen
    Figuren
200 DIM b$(8,8): REM Stellung der Figuren
205 REM Anfangsstellung
210 LET b$(1)="rnbqkbnr"
220 LET b$(2)="pppppppp"
230 LET b$(7)="PPPPPPPP"
240 LET b$(8)="RNBQKBNR"
300 REM Brettdisplay
310 FOR n=1 TO 8: FOR m=1 TO 8
320 LET bc=CODE b$(n,m): INK pw
325 IF bc=CODE"  " THEN GO TO 350: REM Zwischenraum
330 IF bc>CODE " Z " THEN INK pb: LET bc=bc-32: REM klein
    für schwarz
340 LET bc=bc+79: REM in Grafik verwandeln
350 PRINT AT 5+n, 9+m; CHR$ bc
360 NEXT m: NEXT n
400 PAPER 7: INK 0
```

KAPITEL 17

Grafik

Zusammenfassung:
PLOT, DRAW, CIRCLE
POINT
Pixel

In diesem Kapitel sehen wir, wie man mit dem ZX Spectrum zeichnet. Der Bildschirmteil, den Sie dazu verwenden können, hat 22 Zeilen und 32 Spalten, das sind 22*32=704 Zeichenpositionen. Wie Sie aus Kapitel 16 vielleicht noch wissen, besteht jede dieser Zeichenpositionen aus einem Quadrat von 8 mal 8 Punkten. Diese Punkte nennt man *Pixel* (picture elements = Bildelemente).

Ein Pixel wird bestimmt durch zwei Zahlen, seine *Koordinaten*. Die erste, seine *x-Koordinate*, gibt an, wie weit es von der äußersten linken Spalte (merke: x ist ein Kreuz), die zweite, seine *y-Koordinate*, wie weit es von unten entfernt ist. Diese Koordinaten werden in der Regel als Paar in Klammern geschrieben. Demnach sind (0,0), (255,0), (0,175) und (255,175) die Ecken unten links, unten rechts, oben links und oben rechts.

Die Anweisung

PLOT x-Koordinate, y-Koordinate

verleiht dem Pixel mit diesen Koordinaten die angegebene Inkfarbe, so daß dieses Masernprogramm

```
10 PLOT INT (RND*256), INT (RND*176): INPUT a$: GO TO 10
```

jedesmal dann, wenn Sie **ENTER** drücken, einen Zufallspunkt setzt.

Hier ein Programm, das schon interessanter ist. Es zeichnet eine Kurve der Funktion **SIN** (eine Sinuswelle) für Werte zwischen 0 und 2π.

```
10 FOR n=0 TO 255
20 PLOT n,88+80*SIN (n/128*PI)
30 NEXT n
```

Das nächste Programm zeichnet eine Kurve von **SQR** (Teil einer Parabel) zwischen 0 und 4:

```
10 FOR n=0 TO 255
20 PLOT n,80*SQR (n/64)
30 NEXT n
```

Beachten Sie, daß Pixel-Koordinaten sich von Zeile und Spalte in einem **AT**-Posten unterscheiden. Das Schaubild in Kapitel 15 ist Ihnen vielleicht nützlich, wenn Sie Pixel-Koordinaten und Zeilen- und Spaltennummern errechnen wollen.

Der Computer zeichnet, um Ihnen bei Ihren Bildern zu helfen, gerade Linien, Kreise und Teilkreise. Dazu sind die **DRAW**- und **CIRCLE**-Anweisungen da.

Die Anweisung **DRAW** zum Zeichnen einer geraden Linie hat die Form

DRAW x,y

Die Anfangsstelle der Zeile ist das Pixel, wo die letzte **PLOT**-, **DRAW**- oder **CIRCLE**-Anweisung aufgehört hat (man nennt das die **PLOT**-Position). **RUN, CLEAR, CLS** und **NEW** setzen sie zurück an die linke untere Ecke bei (Ø,Ø), die Endstelle befindet sich **x** Pixels rechts davon und **y** Pixels nach oben entfernt. Die **DRAW**-Anweisung allein bestimmt Länge und Richtung der Linie, aber nicht ihren Anfangspunkt.

Experimentieren Sie mit ein paar **PLOT**- und **DRAW**-Anweisungen, etwa mit

```
PLOT Ø,1ØØ: DRAW 8Ø,–35
PLOT 9Ø,15Ø: DRAW 8Ø,–35
```

Merke: Die Zahlen in einer **DRAW**-Anweisung können negativ sein, im Gegensatz zu denen in einer **PLOT**-Anweisung.

Sie können auch in Farbe plotten und zeichnen, müssen dabei aber berücksichtigen, daß Farben stets eine ganze Zeichenposition besetzen und nicht für einzelne Pixel festgelegt werden können. Wenn ein Pixel aufgenommen wird (wie man für Plotten auch sagt), wird es mit der ganzen Inkfarbe besetzt, und die gesamte Zeichenposition übernimmt sie. Das nachstehende Programm zeigt das:

```
1Ø BORDER Ø: PAPER Ø: INK 7: CLS: REM Schirm schwarz
2Ø LET x1=Ø: LET y1=Ø: REM Zeilenanfang
3Ø LET c=1: REM für Inkfarbe, Anfang blau
4Ø LET x2=INT (RND*256): LET y2=INT (RND*176): REM
   Zufallsende der Zeile
5Ø DRAW INK C;x2-x1,y2-y1
6Ø LET x1=x2: LET y1-y2: REM nächste Zeile beginnt, wo die
   letzte aufgehört hat
7Ø LET c=c+1: IF c=8 THEN LET c=1: REM neue Farbe
8Ø GO TO 4Ø
```

Die Zeilen scheinen im Verlauf des Programms breiter zu werden. Das liegt daran, daß eine Zeile die Farben aller mit Ink besetzter Pixel auf allen Zeichenpositionen, die durchlaufen werden, verändert. Zur Beachtung: Sie können **PAPER, INK, FLASH, BRIGHT, INVERSE** und **OVER** in einer **PLOT**- und **DRAW**-Anweisung genauso verwenden wie bei **PRINT** oder **INPUT**. Sie werden zwischen Schlüsselwort und Koordinaten eingefügt und abgeschlossen entweder mit Strichpunkten oder Kommas.

Ein zusätzlicher Vorteil von **DRAW** ist der, daß Sie damit statt gerader Linien Teilkreise zeichnen können. Sie geben dafür eine Zahl zusätzlich ein, um einen Winkel zu bezeichnen, der durchlaufen werden soll. Die Form ist

DRAW x,y,a

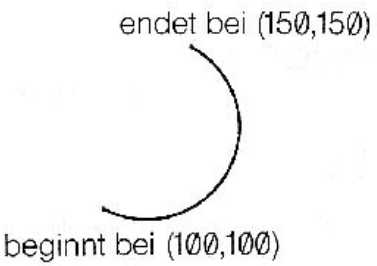
x und **y** geben wie zuvor den Endpunkt der Linie an, **a** ist eine Zahl von Radianten, die unterwegs durchlaufen werden müssen. Ist **a** positiv, geht die Bewegung nach

links, ist es negativ, nach rechts. Man kann **a** auch so sehen, daß es den Teil eines vollständigen Kreises zeigt, der gezeichnet wird: Ein vollständiger Kreis hat 2π Radianten, also zeichnet **a**=π einen Halbkreis, **a**=0.5*π einen Viertelkreis, und so weiter.

Nehmen wir als Beispiel **a**=π. Gleichgültig, welche Werte daneben **x** und **y** haben, es wird ein Halbkreis gezeichnet.
Fahren Sie

1Ø PLOT 1ØØ,1ØØ: DRAW 5Ø,5Ø, PI

Das zeichnet:



Die Zeichnung beginnt in Richtung Südosten, aber bis sie aufhört, geht sie nach Nordwesten. Dazwischen hat sich die Linie um 180 Grad oder π Radianten (Wert von **a**) gedreht.

Fahren Sie das Programm ein paarmal mit anderen Werten für **PI**. Beispiele: **PI/2**, **3*PI/2**, **PI/4** und so weiter.

Die letzte Anweisung, die in diesem Kapitel behandelt wird, ist **CIRCLE**. Damit wird ein ganzer Kreis gezeichnet. Sie geben die Koordinaten des Mittelpunkts und den Radius an, und zwar so

CIRCLE x-Koordinate, y-Koordinate, Radius

Genau wie bei **PLOT** und **DRAW** können Sie die verschiedenen Arten von Farbenposten zu Beginn einer **CIRCLE**-Anweisung einsetzen.

Die Funktion **POINT** sagt Ihnen, ob ein Pixel Ink- oder Paperfarbe hat. Sie besitzt zwei Argumente, die Koordinaten des Pixels (sie müssen in Klammern stehen), und ihr Resultat ist Ø, wenn das Pixel Paperfarbe hat, dagegen 1 bei Inkfarbe. Probieren Sie

CLS: PRINT POINT (Ø,Ø): PLOT Ø,Ø: PRINT POINT (Ø,Ø)

Schreiben Sie

PAPER 7: INK Ø

dann wollen wir uns ansehen, wie **INVERSE** und **OVER** innerhalb einer **PLOT**-Anweisung funktionieren. Die beiden wirken sich nur auf das jeweilige Pixel und nicht auf den Rest der Zeichenpositionen aus. Sie sind in einer **PLOT**-Anweisung normalerweise aus (Ø) und treten einfach dadurch in Aktion (1), daß Sie sie erwähnen.

Hier eine Liste der Möglichkeiten zum Vergleich:

PLOT; –das ist die übliche Form. Damit wird ein Inkpunkt gesetzt, das heißt, das Pixel zeigt dann die Inkfarbe.

PLOT INVERSE 1; – das setzt einen Punkt Tintenlöscher, also zeigt das Pixel dann die Paperfarbe.

PLOT OVER 1; – das verändert das Pixel gegenüber dem vorherigen Zustand. Wenn es Inkfarbe hatte, erhält es Paperfarbe und umgekehrt.

PLOT INVERSE 1; **OVER** 1; – das beläßt das Pixel so, wie es vorher war. Beachten Sie dabei aber auch, daß es die **PLOT**-Position verändert, Sie die Anweisung also dazu verwenden können, eben das zu tun.

Ein weiteres Beispiel für den Gebrauch der **OVER**-Anweisung:
Füllen Sie den Bildschirm mit Geschriebenem schwarz auf weiß und geben Sie dann ein

```
PLOT 0,0: DRAW OVER 1; 225,175
```

Das zeichnet eine recht anständige Linie, die allerdings dort, wo sie auf Geschriebenes trifft, Lücken aufweist. Wiederholen Sie diesen Befehl. Die Zeile verschwindet, ohne irgendeine Spur zu hinterlassen. Das ist der große Vorteil von **OVER 1**. Hätten Sie die Zeile gezeichnet mit

```
PLOT 0,0: DRAW 255,175
```

und gelöscht mit

```
PLOT 0,0: DRAW INVERSE 1; 255,175
```

dann wäre auch von dem Geschriebenen etwas weggelöscht worden.
Nehmen Sie jetzt

```
PLOT 0,0: DRAW OVER 1; 250,175
```

und versuchen Sie zu löschen mit

```
DRAW OVER 1;-250,-175
```

Das funktioniert nicht ganz, weil die Pixel, die die Zeile auf dem Rückweg verwendet, nicht ganz dieselben sind wie diejenigen, die vorher benutzt wurden. Sie müssen eine Zeile genau in derselben Richtung löschen, wie Sie sie gezeichnet haben.

Ein Weg, ungewöhnliche Farben zu erhalten, ist der, zwei vorhandene mit einem vom Benutzer wählbaren Grafikzeichen in einem einzigen Quadrat zusammenzusprenkeln. Fahren Sie dieses Programm:

```
1000 FOR n=0 TO 6 STEP 2
1010 POKE USR "a"+n, BIN 01010101: POKE USR "a"+n+1,
     BIN 10101010
1020 NEXT n
```

Das liefert die vom Benutzer gewählte Grafik, die einem Schachbrettmuster entspricht. Wenn Sie dieses Zeichen (Grafikmodus, dann **a**) in roter Ink- auf gelber Paperfarbe anzeigen, sehe Sie, daß das eine recht annehmbare Orangefarbe ergibt.

Übungen:

1. Spielen Sie mit **PAPER**-, **INK**-, **FLASH**- und **BRIGHT**-Posten in einer **PLOT**-Anweisung. Das sind die Teile, die sich auf die ganze Zeichenposition um das Pixel auswirken. Normalerweise ist das so, als hätte die **PLOT**-Anweisung begonnen mit

```
PLOT PAPER 8: FLASH 8; BRIGHT 8; ...
```

und nur die Inkfarbe einer Zeichenposition wird verändert, wenn dort etwas gesetzt wird, aber wenn Sie wollen, können Sie das ändern.

Seien Sie besonders vorsichtig, wenn Sie bei **INVERSE 1** Farben verwenden, weil das Pixel dadurch auf die Paperfarbe gesetzt, die Inkfarbe aber verändert wird, womit Sie vielleicht nicht rechnen.

2. Verwenden Sie beim Zeichnen von Kreisen **SIN** und **COS** (wenn Sie Kapitel 10 gelesen haben, versuchen Sie herauszufinden, wie). Fahren Sie das Folgende:

```
10 FOR n=0 TO 2*PI STEP PI /180
20 PLOT 100+80*COS n,87+80*SIN n
30 NEXT n
40 CIRCLE 150,87,80
```

Sie können erkennen, daß die **CIRCLE**-Anweisung viel schneller ist, wenn auch weniger exakt.

3. Probieren Sie

```
CIRCLE 100,87,80: DRAW 50,50
```

Das zeigt, daß die **CIRCLE**-Anweisung die **PLOT**-Position an einer recht unbestimmten Stelle verläßt – das ist immer irgendwo die Hälfte der Strecke an der rechten Kreisseite nach oben. Sie müssen in der Regel der **CIRCLE**-Anweisung eine **PLOT**-Anweisung nachschicken, bevor Sie weiterzeichnen.

4. Hier ein Programm, um die Kurve fast jeder Funktion zu zeichnen. Zuerst verlangt es von Ihnen eine Zahl **n**; es setzt die Kurve für Werte von **–n** bis **+n**. Dann wird die Funktion verlangt, eingegeben als String. Der String sollte ein Ausdruck sein, der **x** als das Argument der Funktion verwendet.

```
10 PLOT 0,87: DRAW 255,0
20 PLOT 127,0: DRAW 0,175
30 INPUT s,e$
35 LET t=0
40 FOR f=0 TO 255
50 LET x=(f-128)*s/128: LET y=VAL e$
60 IF ABS y>87 THEN LET t=0: GO TO 100
70 IF NOT t THEN PLOT f,y+88: LET t=1: GO TO 100
80 DRAW 1,y-altes y
100 LET altes y=INT (y+.5)
110 NEXT f
```

Fahren Sie das und geben Sie als Beispiel **10** für die Zahl **n** und **10* TAN x** für die Funktion ein. Das zeichnet eine Kurve von tan x, während x von –10 bis +10 geht.

KAPITEL 18

Bewegung

Zusammenfassung:
PAUSE, INKEY$, PEEK

Sie werden ziemlich oft den Wunsch haben, daß das Programm eine bestimmte Zeit dauert. Dafür ist die **PAUSE**-Anweisung nützlich.

PAUSE n

unterbricht die Computertätigkeit und zeigt *n* Einzelbilder des Fernsehens lang das Bild (bei 50 Einzelbildern pro Sekunde in Europa oder 60 in Amerika). *n* kann bis 65535 gehen. Damit haben Sie knapp unter 22 Minuten; wenn *n* = Ø, heißt das '**PAUSE** ewig'.

Eine Pause kann jederzeit durch Drücken einer Taste beendet werden (beachten Sie, daß ein Leerraum mit **CAPS SHIFT** ebenso zu einer Unterbrechung führt. Sie müssen die Taste drücken, nachdem die Pause angefangen hat.

Dieses Programm betätigt den Sekundenzeiger einer Uhr:

```
1Ø REM Zuerst zeichnen wir das Ziffernblatt
2Ø FOR n=1 TO 12
3Ø PRINT AT 1Ø-1Ø*COS (n/6*PI),16+1Ø*SIN (n/6*PI);n
4Ø NEXT n
5Ø REM Jetzt lassen wir die Uhr laufen
6Ø FOR t=Ø TO 2ØØØØ: REM t ist die Zeit in Sekunden
7Ø LET a=t/3Ø*PI: REM a ist der Winkel des Sekundenzeigers in
   Radianten
8Ø LET sx=8Ø*SIN a: LET sy=8Ø*COS a
2ØØ PLOT 128,88: DRAW OVER 1;sx,sy: REM Sekundenzeiger
   zeichnen
21Ø PAUSE 42
22Ø PLOT 128,88: DRAW OVER 1;sx,sy: REM Sekundenzeiger
   löschen
23Ø NEXT t
```

Diese Uhr läuft wegen Zeile 60 nach rund 55.5 Stunden ab, Sie können sie aber mühelos länger laufen lassen. Sie sehen, wie Zeile 21Ø den Zeitablauf bestimmt. Man möchte erwarten, daß **PAUSE 6Ø** die Uhr mit einem Ticken pro Sekunde laufen läßt, aber das Rechnen erfordert auch seine Zeit und muß berücksichtigt werden. Das geschieht am besten durch Herumprobieren. Man vergleicht die Computeruhr mit einer echten und verändert Zeile 21Ø, bis sie übereinstimmen. (Ganz genau geht das natürlich nicht; eine Anpassung von einem Einzelbild pro Sekunde sind 2% oder eine halbe Stunde am Tag.)

Es gibt einen viel genaueren Weg, die Zeit zu messen. Dabei wird der Inhalt bestimmter Speicherplätze genutzt. Die gespeicherten Daten werden wiedergewonnen durch **PEEK**. Kapitel 25 erläutert im einzelnen, was wir uns hier ansehen. Der verwendete Ausdruck geht so:

(65536*PEEK 23674+256*PEEK 23673+PEEK 23672)/5Ø

Das liefert die Zahl der Sekunden, seitdem der Computer eingeschaltet worden ist (bis hin zu etwa 3 Tagen und 21 Stunden, wo wieder auf Ø geschaltet wird).

Hier ist ein verändertes Uhrenprogramm, mit dem sich das nutzen läßt:

```
10 REM Zuerst zeichnen wir das Ziffernblatt
20 FOR n=1 TO 12
30 PRINT AT 10-10*COS (n/6*PI),16+10*SIN (n/6*PI);n
40 NEXT n
50 DEF FN t()=INT ((65536*PEEK 23674+256*PEEK 23673+
   PEEK 23672)/50): REM Zahl der Sekunden seit Start
100 REM Jetzt lassen wir die Uhr laufen
110 LET t1=FN t()
120 LET a=t1/30*PI: REM a ist der Winkel des Sekundenzeigers
   in Radianten
130 LET sx=72*SIN a: LET sy=72*COS a
140 PLOT 131,91: DRAW OVER 1; sx,sy: REM Zeiger zeichnen
200 LET t=FN t()
210 IF T<=t1 THEN GO TO 200: REM warten bis zur Zeit für den
   nächsten Zeiger
220 PLOT 131,91: DRAW OVER 1; sx,sy: REM alten Zeiger löschen
230 LET t1=t: GO TO 120
```

Die innere Uhr, die diese Methode verwendet, sollte auf ungefähr .Ø1% genau sein, solange der Computer sein Programm fährt; das sind 1Ø Sekunden am Tag. Er hält aber vorübergehend an, sobald Sie mit **BEEP** umgehen, mit dem Kassettenrecorder, den Drucker oder irgendein anderes Zusatzgerät benutzen, das mit dem Computer zu betreiben ist. Sie alle bewirken, daß er Zeit verliert.

Die Zahlen **PEEK 23674**, **PEEK 23673** und **PEEK 23672** sind im Computer gespeichert und werden dazu benützt, in Fünfzigstelsekunden zu zählen. Jede liegt zwischen Ø und 255, und sie vergrößern sich durch alle Zahlen von Ø bis 255 stufenweise; nach 255 fallen sie sofort auf Ø zurück.

Am häufigsten erhöht sich **PEEK 23672**. Jede Fünfzigstelsekunde erfolgt eine Erhöhung um 1. Wenn es bei 255 ist, führt die nächste Erhöhung zu Ø, gleichzeitig schiebt es **PEEK 23673** um 1 höher. Wenn (alle 256 Fünfzigstelsekunden) **PEEK 23673** von 255 auf Ø geschoben wird, erhöht es seinerseits **PEEK 23674** um 1. Das sollte als Erklärung dafür, warum der Ausdruck oben funktioniert, genügen.

Denken Sie nun sorgfältig nach: Angenommen, unsere drei Zahlen sind Ø (für **PEEK 23674**), 255 (für **PEEK 23673**) und 255 (für **PEEK 23672**). Dann sind etwa 21 Minuten seit dem Einschalten vergangen – unser Ausdruck müßte also ergeben

```
(65536*Ø+256*255+255)/5Ø=131Ø.7
```

Aber eine versteckte Gefahr gibt es. Beim nächsten Zähler von 1/5Ø Sekunde verändern sich die drei Zahlen zu 1, Ø und Ø. Das passiert immer wieder, wenn Sie halb damit fertig sind, den Ausdruck auszuwerten: Der Computer würde **PEEK 23674** als Ø bewerten, dann aber die beiden anderen auf Ø verändern, bevor es sie mit Peek erreichen kann. Die Antwort wäre dann

(65536*Ø+256*Ø+Ø)/5Ø=Ø

was natürlich überhaupt nicht stimmt.

Eine einfache Regel, um das zu vermeiden: Man verwendet den Ausdruck zweimal hintereinander und nimmt die größere Antwort.

Wenn Sie sich das obige Programm genau ansehen, wird Ihnen auffallen, daß es das stillschweigend miterledigt.

Hier ein Trick zur Anwendung der Regel. Definieren Sie Funktionen

```
1Ø DEF FN m(x,y)=(x+y+ABS (x-y))/2: REM der größere Wert von
   x und y
2Ø DEF FN u()=(65536*PEEK 23674+256*PEEK 23673+ PEEK
   23672)/5Ø: REM Zeit, kann falsch sein
3Ø DEF FN t()=FN m(FN u(),FN u()): REM Zeit, richtig
```

Sie können die drei Zählerzahlen so verändern, daß sie nicht die abgelaufene Zeit seit dem Einschalten des Computers, sondern die richtige Zeit anzeigen. Beispiel: Um die Zeit auf 1Ø.ØØ morgens festzusetzen, errechnen Sie, daß das 1Ø*6Ø*6Ø*5Ø=18ØØØØ Fünfzigstelsekunden sind, und daß

18ØØØØ=65536*27+256*119+64

Um die drei Zahlen auf 27, 119 und 64 zu setzen, schreiben Sie

```
POKE 23674,27: POKE 23673,119: POKE 23672,64
```

In Ländern mit Stromversorgungsfrequenzen von 6Ø Hertz müssen Sie in diesen Programmen dort, wo nötig, '5Ø' durch '6Ø' ersetzen.

Die Funktion **INKEY$** (die kein Argument hat) liest die Tastatur. Wenn Sie exakt eine Taste drücken (oder eine **SHIFT**-Taste und dazu nur eine einzige Taste), ist das Resultat das Zeichen, das diese Taste im L-Modus liefert; im übrigen ist das Resultat der leere String.

Probieren Sie dieses Programm aus, das wie eine Schreibmaschine funktioniert.

```
1Ø IF INKEY$ <> " " THEN GO TO 1Ø
2Ø IF INKEYS = " " THEN GO TO 2Ø
3Ø PRINT INKEY$;
4Ø GO TO 1Ø
```

Zeile 1Ø wartet hier darauf, daß Sie den Finger von der Tastatur nehmen, Zeile 2Ø darauf, daß Sie eine neue Taste drücken.

Denken Sie daran, daß **INKEY$** im Gegensatz zu **INPUT** nicht auf Sie wartet. Sie drücken also nicht auf **ENTER**, aber wenn Sie gar nichts drücken, haben Sie Ihre Chance verpaßt.

Übungen

1. Was passiert, wenn Sie Zeile 10 im Schreibmaschinenprogramm weglassen?

2. **INKEY$** kann auch zusammen mit **PAUSE** verwendet werden, wie in diesem zweiten Schreibmaschinenprogramm:

```
10 PAUSE 0
20 PRINT INKEY$;
30 GO TO 10
```

Warum ist es, damit das klappt, notwendig, daß eine Pause nicht aufhört, wenn Sie bei Beginn des Programms schon eine Taste drücken?

3. Verändern Sie das zweite Uhrzeigerprogramm so, daß es auch Minuten- und Stundenzeiger liefert und sie jede Minute einmal zeichnet. Wenn Sie Ehrgeiz verspüren: Richten Sie es so ein, daß jede Viertelstunde irgendeine Vorführung stattfindet – Sie könnten mit **BEEP** den Westminsterschlag erzeugen. (Siehe das nächste Kapitel.)

4. (Etwas für Sadisten.) Wie wär's damit:

```
10 IF INKEY$="" THEN GO TO 10
20 PRINT AT 11,14; "AUTSCH!"
30 IF INKEY$ <>"" THEN GO TO 30
40 PRINT AT 11,14; "      "
50 GO TO 10
```

KAPITEL 19

BEEP

Zusammenfassung:
BEEP

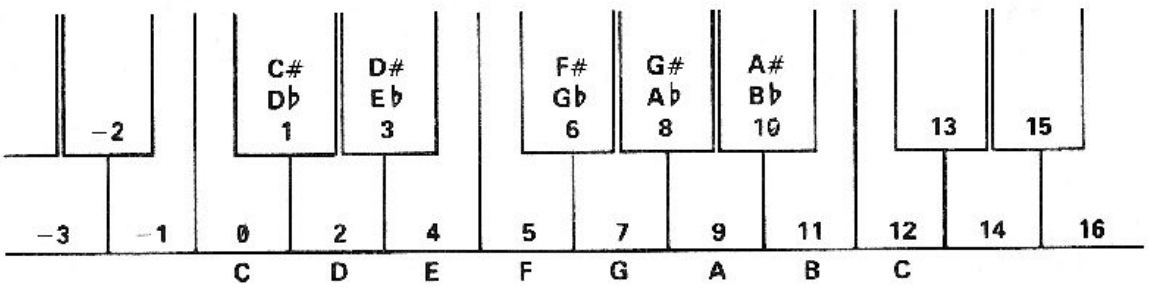
Wenn Sie nicht schon entdeckt haben, daß im ZX Spectrum ein Lautsprecher eingebaut ist, lesen Sie das Anleitungsheft, bevor Sie weitermachen.

Der Lautsprecher wird zum Tönen gebracht durch die **BEEP**-Anweisung

BEEP Zeitdauer, Tonhöhe

wobei wie gewohnt 'Zeitdauer' und 'Tönhöhe' durch beliebige numerische Ausdrücke bestimmt werden. Die Zeitdauer wird in Sekunden angegeben, die Tonhöhe durch Halbtöne über dem mittleren C – für Töne darunter mit negativen Zahlen.

Hier eine Zeichnung mit den Höhenwerten aller Töne in einer Oktave auf dem Klavier:



Wenn Sie höhere oder tiefere Töne haben wollen, müssen Sie für jede Oktave nach oben oder unten 12 dazulegen oder wegtun.

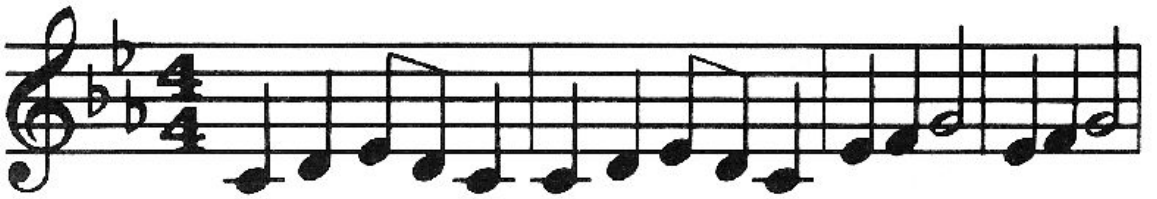
Falls Sie ein Klavier vor sich haben, wenn Sie eine Melodie programmieren, brauchen Sie wahrscheinlich nicht mehr als diese Zeichnung, um die Höhenwerte festzulegen. Wenn Sie aber direkt von Noten abschreiben, schlagen wir vor, daß Sie eine Zeichnung von der Strophe anfertigen, an jeder Note den Wert der Tonhöhe vermerken und die Tonart berücksichtigen.

Schreiben Sie beispielsweise

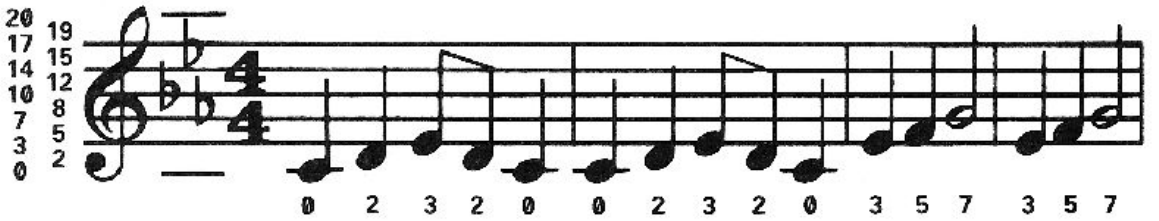
```
10 PRINT "Frere Gustav"
20 BEEP 1,0: BEEP 1,2: BEEP .5,3: BEEP .5,2: BEEP 1,0
30 BEEP 1,0: BEEP 1,2: BEEP .5,3: BEEP .5,2: BEEP 1,0
40 BEEP 1,3: BEEP 1,5: BEEP 2,7
50 BEEP 1,3: BEEP 1,5: BEEP 2,7
60 BEEP .75,7; BEEP .25,8: BEEP .5,7: BEEP .5,5: BEEP .5,3:
   BEEP .5,2: BEEP 1,0
70 BEEP .75,7: BEEP .25,8: BEEP .5,7: BEEP .5,5: BEEP .5,3:
   BEEP .5,2: BEEP 1,0
80 BEEP 1,0: BEEP 1,-5: BEEP 2,0
90 BEEP 1,0: BEEP 1,-5: BEEP 2,0
```

Wenn Sie das fahren, müßte der Trauermarsch aus Gustav Mahlers 1. Symphonie herauskommen, der Teil, wo die Kobolde den amerikanischen Reiter begraben.

Nehmen wir an, Ihre Melodie wäre, wie die Mahlersymphonie, in c-Moll geschrieben. Der Anfang sieht so aus:



und Sie können die Werte für die einzelnen Noten so eintragen:



Wir haben noch zwei Hilfslinien gezogen, damit man sich besser auskennt. Beachten Sie, wie das Es in der Vorzeichnung nicht nur das E ganz oben beeinflußt und es von 16 auf 15 erniedrigt, sondern auch das E auf der untersten Linie, das von 4 auf 3 erniedrigt wird. Von jetzt an müßte es eigentlich ganz einfach sein, den Höhenwert jeder Note zu finden.

Wenn Sie die Tonart ändern wollen, setzen Sie am besten eine Variable **ton** (für Tonart) und geben vor jedem Höhenwert **ton+** ein. Dann wird aus der zweiten Zeile:

```
20 BEEP 1, ton+0: BEEP 1,ton+2: BEEP .5,ton+3: BEEP .5,ton+2:
   BEEP 1,ton+0
```

Bevor Sie ein Programm fahren, müssen Sie **ton** den passenden Wert – 0 für c-moll, 2 für d-moll, 12 für c-moll eine Oktave höher – geben, und so weiter. Sie können den Computer durch die Anpassung von **ton** auf ein anderes Instrument mit Bruchwerten einstimmen.

Auch die Dauer aller Töne müssen Sie festlegen. Da das ein ziemlich langsames Musikstück ist, haben wir für eine viertel Note eine Sekunde genommen und den Rest entsprechend festgelegt, eine halbe Sekunde für ein Achtel, und so weiter.

Beweglicher ist man, wenn man eine Variable **viertel** setzt, um die Länge einer Viertelnote zu speichern und die Zeitdauer jeweils danach zu bestimmen. Dann sähe Zeile 20 so aus:

```
20 BEEP viertel,ton+0: BEEP viertel,ton+2: BEEP viertel,ton+0:
   BEEP viertel/2,ton+3:BEEP viertel/2,ton+2:BEEP viertel,ton+0
```

Wenn Sie **viertel** die entsprechenden Werte geben, können Sie das Musikstück mühelos beschleunigen.

Versuchen Sie, selbst Melodien zu programmieren. Fangen Sie mit einfachen an, etwa mit 'Hänschen klein'. Sollten Sie weder ein Klavier noch Notenblätter haben, besorgen Sie sich ein ganz einfaches Instrument wie eine Blockflöte, und stellen Sie Ihre Melodien damit zusammen. Sie können eine Zeichnung machen, die den Höhenwert für jeden Ton angibt, den Sie mit diesem Instrument spielen können.

Schreiben Sie:

```
FOR n=Ø TO 1ØØØ: BEEP .5,n: NEXT n
```

Das spielt die Tonleiter hinauf, so hoch es geht, und hört dann mit der Fehlermeldung **B integer out of range** auf. Sie können n anzeigen lassen, um zu erfahren, wie hoch der Computer wirklich gekommen ist.

Versuchen Sie das auch umgekehrt. Die tiefsten Töne werden nur noch Klicklaute sein; übrigens bestehen auch die höheren Töne aus solchen Klicklauten, nur geht das schneller, so daß das Ohr sie nicht unterscheiden kann.

Für gute Musik ist nur der mittlere Tonbereich sinnvoll; die tiefen Töne klingen zu sehr nach einem Klicken, die hohen sind dünn und tremolieren ein bißchen:

Geben Sie diese Programmzeile ein:

```
1Ø BEEP .5,Ø: BEEP .5,2: BEEP .5,4: BEEP .5,5: BEEP .5,7:
   BEEP .5,9: BEEP .5,11: BEEP .5,12: STOP
```

Das spielt die Tonleiter in C-Dur, also alle weißen Tasten auf dem Klavier vom mittleren C bis zum nächsten. Gestimmt ist diese Tonleiter genau wie auf dem Klavier. Man nennt das diatonisch, weil die Tonintervalle gleich bleiben. Ein Geiger würde die Tonleiter ein wenig anders spielen und alle Töne leicht verändern, damit sie eingängiger sind. Das bringt er dadurch zustande, daß er die Finger an der Seite ganz wenig auf- und abgleiten läßt, was ein Pianist natürlich nicht tun kann.

Die natürliche Tonleiter, die der Geiger spielt, klingt dann so:

```
2Ø BEEP .5,Ø: BEEP .5,2.Ø39: BEEP .5,3.86: BEEP .5,4.98:
   BEEP .5,7.Ø2: BEEP .5,8.84: BEEP .5,1Ø.88: BEEP .5,12: STOP
```

Kann sein, daß Sie einen Unterschied zwischen beiden entdecken oder auch nicht: manche Leute können es. Der erste wesentliche Unterschied ist der, daß der dritte Ton in der natürlichen Tonleiter ein wenig erniedrigt ist. Falls Sie ein echter Perfektionist sein sollten, wollen Sie Ihre Melodien vielleicht nach dieser natürlichen Tonleiter programmieren. Der Nachteil: In C geht das zwar recht gut, aber in anderen Tonarten weniger, weil sie alle ihre eigene natürliche Tonleiter haben; bei manchen Tonarten wird das sogar ausgesprochen häßlich. Die diatonische Tonleiter unterscheidet sich nur wenig und klingt in allen Tonarten gleich gut.

Beim Computer ist das natürlich weniger ein Problem, weil Sie den Kniff verwenden können, eine Variable **ton** zu setzen.

Bei mancher Musik – vor allem bei der indischen – werden Tonintervalle genutzt, die kleiner sind als ein Halbton. Sie können das ohne Schwierigkeiten in die **BEEP**-Anweisung einbauen; so hat etwa der Viertelton über dem mittleren C den Wert 5.

Sie können die Tastatur piepen statt klicken lassen durch Eingabe von

POKE 23609,255

Die zweite Zahl hier bestimmt die Länge des Tons. Probieren Sie verschiedene Töne zwischen 0 und 255 aus. Bei 0 ist der Ton so leise, daß er wie ein schwaches Klicken klingt.

Wenn Sie daran interessiert sind, mit dem Ton beim ZX Spectrum mehr zu machen, etwa den Ton zu hören, den **BEEP** anderswo macht, als im eingebauten Lautsprecher, werden Sie feststellen, daß das Signal in den 'Ear'- und 'Mic'-Buchsen ebenfalls vorhanden ist. In der 'Ear'-Buchse ist er höher, aber sonst sind sie gleich. Sie können das nutzen, wenn Sie einen Ohrstecker oder Kopfhörer an Ihren Spectrum anschließen. Der eingebaute Lautsprecher wird dadurch nicht abgeschaltet. Wenn Sie es wirklich darauf anlegen wollen, eine Menge Lärm zu machen, könnten Sie einen Verstärker anschließen – die 'Mic'-Buchse liefert wohl ungefähr den richtigen Wert – oder Sie können die Töne auf Band nehmen und den Spectrum veranlassen, mit sich im Takt zu spielen.

Sie beschädigen den Spectrum selbst dann nicht, wenn Sie die 'Mic'- oder 'Ear'-Buchsen kurzschließen. Experimentieren Sie also unbesorgt, um das zu erhalten, was Sie brauchen.

Übung:

1. Schreiben Sie das Mahler-Programm so um, daß es **FOR**-Schleifen dazu benützt, die Takte zu wiederholen.

Programmieren Sie den Computer so, daß er nicht nur den Trauermarsch spielt, sondern auch den Rest von Mahlers Symphonie.

KAPITEL 20

Speichern auf Tonband

Zusammenfassung:
LOAD, SAVE, VERIFY, MERGE

Die Grundkenntnisse dafür, wie man mit dem Kassettenrecorder Programme auf Band nimmt, wieder in den Computer lädt und bestätigt (**SAVE, LOAD** und **VERIFY**), erwerben Sie im Anleitungsheft. Diesen Abschnitt sollten Sie auf jeden Fall lesen und die Prozedur praktisch ausprobieren, bevor Sie sich hier auf die Sache näher einlassen.

Wir haben gesehen, daß **LOAD** das alte Programm und vorhandene Variablen im Computer löscht, bevor das neue mit neuen Variablen vom Band geladen wird. Es gibt noch eine Anweisung, nämlich **MERGE**, die das nicht tut. **MERGE** löscht alte Programme und Variablen nur dann, wenn es muß, weil das Neue dieselbe Zeilennummer oder denselben Namen hat. Geben Sie das 'Würfel'-Programm von Kapitel 11 ein und sichern Sie es unter dem Namen "Würfel" auf Band. Dann tippen Sie das folgende:

```
 1 PRINT 1
 2 PRINT 2
1Ø PRINT 1Ø
2Ø LET x=2Ø
```

und machen Sie wie bei der üblichen Prüfung weiter, ersetzen **VERIFY "Würfel"** aber durch

```
MERGE "Würfel"
```

Wenn Sie das Programmlisting holen, sehen Sie, daß die Zeilen 1 und 2 unversehrt geblieben sind, die Zeilen 1Ø und 2Ø aber durch die Zeilen mit denselben Nummern aus dem Würfelprogramm ersetzt worden sind. **x** hat also überlebt (versuchen Sie **PRINT x**).

Sie haben jetzt einfache Formen der vier Anweisungen gesehen, die bei Bandkassetten verwendet werden:

SAVE zeichnet das Programm und die Variablen auf Band auf.

VERIFY vergleicht das Programm und die Variablen auf der Kassette mit denen, die sich jetzt im Computer befinden.

LOAD löscht im Computer alle Programme und Variablen und ersetzt sie durch neue, die von der Kassette eingelesen werden.

MERGE ist wie **LOAD**, löscht eine alte Programmzeile oder Variable aber nur dann, wenn es sein muß, weil das Programm den selben Namen hat oder Zeilennummern im alten und neuen Programm identisch sind.

In allen Fällen folgt dem Schlüsselwort ein String. Bei **SAVE** liefert er einen Namen für das Programm auf Band, während er bei den drei anderen dem Computer angibt, welches Programm er suchen soll. Während er sucht, zeigt er auf dem Bildschirm den Namen jedes Programms an, auf das er stößt. Bei den Vorgängen gibt es noch ein paar Eigenheiten.

Bei **VERIFY, LOAD** und **MERGE** können Sie den leeren String als Suchnamen

liefern. Dann ist dem Computer der Name egal, er nimmt das erste Programm, das ihm über den Weg kommt.

Eine Abart von **SAVE** hat die Form

SAVE String **LINE** Nummer

Ein Programm, das man auf diese Weise sichert, wird so aufgenommen, daß es, wenn es bei **LOAD** (aber nicht bei **MERGE**) zurückgelesen wird, automatisch zu der angegebenen Zeile springt, also von selbst fährt.

Bis jetzt haben wir auf Kassette nur Programme gemeinsam mit ihren Variablen gespeichert. Es gibt auch noch zwei andere Arten mit den Namen *Arrays* und *Bytes.*

Arrays werden ein wenig anders behandelt:

Sie können Arrays dadurch auf Band speichern, daß Sie in einer **SAVE**-Anweisung **DATA** verwenden, und zwar so

SAVE String DATA Arrayname()

String ist der Name, den die Information auf Band tragen wird; das funktioniert genauso wie beim Sichern eines Programms oder schlichter Bytes.

Der Arrayname benennt das Array, das Sie sichern wollen, ist also nur ein Buchstabe (Variable) oder ein Buchstabe mit $ (String). Denken Sie an die Klammern danach; Sie mögen der Meinung sein, daß sie von der Logik her unnötig wären, müssen sie aber trotzdem einsetzen, um es dem Computer leichter zu machen.

Seien Sie sich klar über die verschiedenen Rollen von *String* und *Arrayname*. Wenn Sie z.B. sagen

SAVE "Knacke" DATA b()

dann holt **SAVE** das Array *b* aus dem Computer und speichert es unter dem Namen „Knacke" auf Band. Wenn Sie schreiben

VERIFY "Knacke" DATA b()

sucht der Computer nach einem Zahlenarray, das unter dem Namen "Knacke" auf Band gespeichert ist (wenn es ihn findet, zeigt er an 'Number array: Knacke') und vergleicht es mit dem Array *b* im Computer.

LOAD "Knacke" DATA b()

findet das Array auf Band, löscht dann – falls im Computer noch Platz dafür ist – jedes schon vorhandene Array namens *b* und lädt das neue Array vom Band, das nun den Namen "b" hat.

Bei gesicherten Arrays können Sie **MERGE** nicht verwenden.

Sie können Zeichen-(String-)Arrays auf genau dieselbe Weise sichern. Wenn der Computer das Band absucht und eines davon findet, zeigt er an 'character array:' gefolgt vom Namen. Wenn Sie ein Zeichenarray laden, löscht es nicht nur jedes schon vorhandene Zeichenarray mit demselben Namen, sondern auch jeden String mit demselben Namen.

Bytespeicherung wird dazu benützt, Einzelinformationen ohne Hinweis darauf zu speichern, wofür sie verwendet werden sollen – das kann ein Fernsehbild sein oder vom Benutzer gewählte Grafik oder etwas, das Sie selbst erfunden haben. Anzeigen kann man mit dem Wort **CODE**, wie in

SAVE "picture" CODE 16384,6912

Die Speicherungseinheit im Speicher ist das *Byte* (eine Zahl zwischen Ø und 255), und jedes Byte hat eine *Adresse* (eine Zahl zwischen Ø und 65535). Die erste Zahl nach **CODE** ist die Adresse des ersten Bytes, das auf Band gespeichert werden soll, die zweite die Zahl der zu speichernden Bytes. In unserem Fall ist 16384 die Adresse des ersten Bytes in der Displaydatei (die das Fernsehbild enthält), 6912 die Zahl der darin enthaltenen Bytes, so daß wir eine Kopie des Fernsehbilds speichern – probieren Sie das aus. Der Name **"picture"** wirkt genau wie die Namen für Programme.

Um zurückzuladen, verwenden Sie

LOAD "picture" CODE

Sie können nach **CODE** Zahlen setzen, und zwar in der Form

LOAD Name **CODE** Anfang, Länge

Länge ist hier nur eine Sicherheitsmaßnahme. Wenn der Computer die Bytes mit dem richtigen Namen auf dem Band gefunden hat, lädt er sie trotzdem nicht, wenn es mehr sind als *Länge* – da offenkundig mehr Daten vorhanden sind, als Sie erwartet haben, könnte er sonst etwas überschreiben, das Sie nicht überschrieben haben wollen. Er bringt die Fehlermeldung **R Tape loading error** (Fehler beim Laden auf Band). Sie können *Länge* weglassen, dann liest der Computer die Bytes ohne Rücksicht darauf ein, wie viele es sind.

Anfang zeigt die erste Adresse an, wohin das erste Byte zurückgeladen werden soll – sie kann sich unterscheiden von der Adresse, aus der es gesichert worden ist. Wenn sie gleich sind, können Sie Anfang in der *LOAD*-Anweisung auch weglassen.

CODE 16384,6912 ist für Sichern und Laden des Bildes so nützlich, daß Sie es durch **SCREEN$** ersetzen können, etwa bei

SAVE "picture" SCREEN$
LOAD "picture" SCREEN$

Das ist einer der seltenen Fälle, bei denen **VERIFY** nicht funktioniert. **VERIFY** zeigt die Namen des auf Band Gefundenen an, und bis es zum Vergleich kommt, hat die Displaydatei sich verändert, so daß eine Bestätigung nicht erfolgen kann. In allen anderen Fällen sollten Sie bei jeder Verwendung von **SAVE** auch **VERIFY** verwenden.

Unten folgt eine vollständige Zusammenfassung der vier in diesem Kapitel verwendeten Anweisungen.

Name steht für jeden Stringausdruck und bezieht sich auf den Namen, unter dem die Information auf Kassette gespeichert ist. Er sollte aus ASCII-Anzeigezeichen bestehen, von denen nur die ersten 1Ø benützt werden.

Es gibt vier Arten von Information, die man auf Band speichern kann: Programm und Variable (gemeinsam), Zahlenarrays, Zeichenarrays und nackte Bytes.

Wenn **VERIFY, LOAD** und **MERGE** das Band nach Informationen mit einem gegebenen Namen und einer gegebenen Art absuchen, zeigen sie auf dem Bildschirm Art und Namen jeder gefundenen Information an. Die Art wird mitgeteilt durch 'Program:' 'Number array:' 'Character array:' oder 'Bytes:'. Wenn *Name* der leere String war, nehmen sie die erste Informationsmenge der richtigen Art, ohne Rücksicht auf den Namen.

SAVE

Sichert Information auf Band unter dem gegebenen Namen. Fehler F tritt auf, wenn Name leer ist oder 11 und mehr Zeichen hat.

SAVE liefert stets die Meldung **Start tape, press any key** (Band starten und eine beliebige Taste drücken) und wartet, bis eine Taste gedrückt worden ist, bevor etwas gesichert wird.

1. Für Programm und Variable ist die Normalform:

SAVE Name

Die Form

SAVE Name **LINE** Zeilennummer

sichert das Programm und die Variablen auf solche Weise, daß nach Ausführung von **LOAD** automatisch folgt

GO TO Zeilennummer

SAVE Name **LINE** entspricht **SAVE** Name **LINE 1**. Wenn das Programm geladen wird, fährt es demnach von Anfang selbst.

2. Bytes:

SAVE Name **CODE** Anfang, Länge

sichert *Länge* Bytes, beginnend bei Adresse *Anfang*.

SAVE Name **SCREEN$**

entspricht

SAVE Name **CODE 16384,6912**

und sichert das Bild vom Fernsehschirm.

3. Arrays:

SAVE Name **DATA** Buchstabe ()

oder

SAVE Name **DATA** Buchstabe $ ()

sichert das Array mit dem Namen *Buchstabe* oder *Buchstabe $* (eine Beziehung zu *Name* braucht nicht zu bestehen).

VERIFY

Vergleicht die Information auf Band mit der schon gespeicherten Information. Wenn die Prüfung nicht möglich ist, erscheint die Fehlermeldung **R tape loading error.**

1. Programm und Variable:

VERIFY Name

2. Bytes:

VERIFY Name **CODE** Anfang, Länge

Ist die Zahl der Bytes *Name* auf Band größer als *Länge*, kommt Fehlermeldung R. Sonst wird mit den Bytes im Speicher verglichen, beginnend bei Adresse *Anfang*.

VERIFY Name **SCREEN$**

entspricht

VERIFY Name **CODE 16384,6912**

wird aber fast mit Sicherheit nicht prüfen.

3. Arrays:

LOAD Name **DATA** Buchstabe ()

oder

LOAD Name **DATA** Buchstabe $ ()

löscht jedes Array, das schon *Buchstabe* oder *Buchstabe $* heißt, und bildet aus dem auf Kassette gespeicherten Array ein neues.

Fehler **4 Out of memory** (kein Speicherplatz mehr) tritt auf, wenn für neue Arrays kein Platz mehr ist. Alte Arrays werden nicht gelöscht.

MERGE

Lädt neue Information von der Kassette, ohne alte Information im Speicher zu löschen.

1. Programm und Variable:

MERGE Name

mischt das Programm *Name* mit dem schon im Speicher vorhandenen und über-

schreibt alle Programmzeilen oder Variablen im alten Programm, deren Zeilennummern oder Namen mit solchen im neuen Programm übereinstimmen.

Fehler **4 Out of memory** tritt auf, wenn im Speicher nicht mehr genug Platz ist für das ganze alte Programm und die Variablen *und* das ganze neue Programm und die Variablen, die neu vom Band geladen werden.

2. Bytes:

Nicht möglich

3. Arrays:

Nicht möglich

Übungen:

1. Stellen Sie eine Kassette her, auf der das erste Programm beim Laden ein *Menü* anzeigt (eine Liste der anderen Programme auf der Kassette), Sie auffordert, ein Programm zu wählen, und es dann lädt.

2. Geben Sie die Schachfiguren-Grafikzeichen von Kapitel 14 ein und drücken Sie **NEW** (die überstehen das). Das Abschalten des Computers überleben sie allerdings nicht. Wenn Sie sie behalten wollen, müssen Sie sie auf Band sichern und dabei **SAVE** mit **CODE** verwenden. Am einfachsten sichert man alle einundzwanzig benutzergewählte Grafikzeichen durch

SAVE "Schach" CODE USR "a", 21*8

gefolgt von

VERIFY "Schach" CODE

Das ist das System der *Byte*-Sicherung, die dazu benützt wurde, das Bild sicherzustellen. Die Adresse des ersten Bytes, das gesichert werden soll, ist **USR "a"**, die Adresse des ersten der acht Bytes, die das Muster des ersten benutzergewählten Grafikzeichens bestimmen, und die Zahl der zu sichernden Bytes ist 21*8 – für jedes der 21 Grafikzeichen acht Bytes.

Zum Zurückladen würden Sie normalerweise

LOAD "Schach" CODE

verwenden. Wenn Sie aber mit einer anderen Speichermenge in einen Spectrum zurückladen oder die benutzergewählten Grafikzeichen an eine andere Adresse gestellt haben (das muß man bewußt tun, und zwar mit einer komplizierteren Methode), müssen Sie vorsichtiger sein und

LOAD "Schach" CODE USR "a"

benützen. **USR** berücksichtigt, daß die Grafikzeichen an eine andere Adresse zurückgeladen worden sein könnten.

KAPITEL 21

Der ZX-Drucker

Zusammenfassung:
LPRINT, LLIST, COPY
Merke: Keiner dieser Befehle entspricht der BASIC-Norm, allerdings wird **LPRINT** auch von einigen anderen Computern verwendet.

Wenn Sie einen ZX-Drucker haben, werden Sie dazu eine Anleitung bekommen haben. Das folgende Kapitel befaßt sich mit den BASIC-Anweisungen, die man benötigt, damit er funktioniert.

LPRINT und **LLIST** sind genau wie **PRINT** und **LIST**, nur verwenden sie statt des Fernsehapparats den Drucker. (Das L ist übrigens ein historischer Zufall. Als BASIC erfunden wurde, benutzte man anstelle eines Fernsehers meistens eine elektrische Schreibmaschine, so daß **PRINT** das ja 'Drucken' heißt, auch wirklich Drucken bedeutete. Wenn man mengenweise Protokolle haben wollte, verwendete man einen sehr schnellen Zeilen- oder Paralleldrucker (englisch: line printer), und eine **LPRINT**-Anweisung bedeutete: 'Line printer **PRINT**'.)

Probieren Sie einmal das folgende Programm aus:

```
10 LPRINT "Dieses Programm"
30 LPRINT "druckt den Zeichenvorrat:"
40 FOR n=32 TO 255
50 LPRINT CHR$ n;
60 NEXT n
```

Die dritte Anweisung **COPY** druckt eine Kopie des Fernsehbilds. Beispiel: Drücken Sie **LIST**, um ein Listing des obigen Programms auf den Bildschirm zu bekommen, und geben Sie

```
COPY
```

ein. Beachten Sie, daß **COPY** nicht bei einem der Listings funktioniert, die der Computer automatisch liefert, weil das verschwindet, sobald ein Befehl ausgeführt wird. Sie müssen in diesem Fall vorher **LLIST** verwenden und **COPY** sein lassen.

Sie können den laufenden Drucker jederzeit anhalten mit einem Druck auf **BREAK** (**CAPS SHIFT** und **SPACE**).

Wenn Sie diese Befehle ohne angeschlossenen Drucker ausführen lassen, müßte die ganze Ausgabe verlorengehen und der nächste Befehl befolgt werden.

Probieren Sie aus:

```
10 FOR n=31 TO 0 STEP-1
20 PRINT AT 31-n,n; CHR$ (CODE "0"+n);
30 NEXT n
```

Sie werden ein Muster von Zeichen sehen, das diagonal von der oberen rechten Ecke herunterläuft, bis es unten am Bildschirm ankommt, wo es mit der Fehlermeldung "*out of screen*" (außerhalb des Bildschirmes) abschließt.

Verändern Sie nun **AT 31–n,n** in Zeile 20 zu **TAB n.** Das Programm wird genau dieselbe Wirkung haben wie vorher, aber fragen, ob es abrollen soll ("**scroll?**").

Machen Sie dann aus **PRINT** in Zeile 20 **LPRINT**. Diesmal erscheint kein **scroll?**, da es beim Drucker nicht vorkommt, und das Muster wird bis zum Ende weiterlaufen.

Verändern Sie jetzt **TAB n** zu **AT 31–n,n**, immer noch mit **LPRINT**. Diesmal erhalten Sie nur eine Einzelzeile von Zeichen. Der Grund für den Unterschied: Die Ausgabe von **LPRINT** wird nicht sofort gedruckt, sondern ordnet in einem Pufferspeicher ein Bild von einer Zeilenlänge ein, das umfaßt, was der Computer dem Drucker übermitteln wird, wenn er dazu kommt. Das Drucken findet statt

(I) wenn der Puffer voll ist,

(II) nach einer **LPRINT**-Anweisung, die nicht mit einem Komma oder einem Strichpunkt endet,

(III) wenn ein Komma, ein Apostroph oder ein **TAB**- Posten eine neue Zeile verlangt, oder

(IV) am Ende des Programms, wenn noch etwas ungedruckt geblieben ist.

(III) erklärt, warum unser Programm mit **TAB** gerade so funktioniert. Was **AT** angeht, bleibt die Zeilennummer unbeachtet, und die **LPRINT**-Position wird (wie die **PRINT**-Position, aber für den Drucker statt des Fernsehers) zur Spaltennummer verändert. Ein **AT**-Posten kann nie bewirken, daß eine Zeile zum Drucker geschickt wird.

Übung:

1. Erzeugen Sie eine gedruckte Kurve von **SIN** mit dem Programm in Kapitel 17 und der anschließenden Verwendung von **COPY**.

KAPITEL 22

Andere Peripheriegeräte

Es gibt noch mehr Geräte, die Sie an den Spectrum anschließen können.

Der ZX Mikrodrive (Diskettenlaufwerk) ist ein schnelles Großspeichergerät und in der Anwendung weitaus vielseitiger als ein Kassettenrecorder. Es ist nicht nur mit **SAVE**, **VERIFY**, **LOAD** und **MERGE** zu betreiben, sondern auch mit **PRINT**, **LIST**, **INPUT** und **INKEY$**.

Das Netz wird dazu verwendet, mehrere Spectrum-Computer anzuschließen, so daß sie miteinander reden können – ein Nutzen dabei ist der, daß Sie nur ein Laufwerk für mehrere Computer brauchen.

Das Interface RS232 ist ein Normanschlußgerät, mit dem Sie einen Spectrum an Tastaturen, Drucker, Computer und verschiedene andere Maschinen anschließen können, selbst wenn sie nicht eigens für den Spectrum gebaut worden sind.

Sie verwenden zum Teil zusätzliche Schlüsselwörter, die auf der Tastatur stehen, aber nicht ohne die Zusatzgeräte genutzt werden können: Das sind **OPEN#, CLOSE#, MOVE, ERASE, CAT** und **FORMAT**.

KAPITEL 23

IN und **OUT**

Zusammenfassung:
IN
OUT

Der Prozessor kann mit **PEEK** und **POKE** im Speicher lesen und (zumindest in den RAM) etwas hineinschreiben. Dem Prozessor selbst ist es gleichgültig, ob der Speicher ROM, RAM oder überhaupt nichts ist; er weiß nur, daß es 65536 Speicheradressen gibt, und er kann ein Byte von jedem lesen (auch wenn es Unsinn ist) und jedem ein Byte eingeben (auch wenn es verlorengeht). Auf ganz gleiche Weise gibt es 65536 sogenannte *I/O-Ports*, das sind Ein/Ausgabe-Elemente. Der Prozessor benützt sie dazu, mit Vorrichtungen wie der Tastatur oder dem Drucker Verbindung zu halten. Über BASIC können sie mit der **IN**-Funktion und der **OUT**-Anweisung gesteuert werden.

IN ist eine Funktion wie **PEEK.**

IN Adresse

Sie besitzt ein Argument, die Adresse des Eingabebausteins, ihr Resultat ist ein Byte, das aus diesem Baustein gelesen wird.

OUT ist eine Anweisung wie **POKE**.

OUT Adresse, Wert

schreibt den gegebenen Wert in den Baustein mit der genannten Adresse. Wie die Adresse interpretiert wird, hängt sehr vom Computer als solchem ab; recht oft bedeuten viele verschiedene Adressen dasselbe. Beim Spectrum ist es am vernünftigsten, sich die Adresse binär geschrieben vorzustellen, weil die einzelnen Bits dazu neigen, sich unabhängig voneinander zu verhalten. Es gibt 16 Bits, die wir (unter Verwendung von A für *Adresse*) nennen wollen

A15, A14, A13, A12,....., A2, A1, A0

A0 ist hier das 1er-Bit, A1 das 2er-Bit, A3 das 4er-Bit, und so weiter. Die wichtigen Bits sind A0, A1, A2, A3 und A4. Sie sind normalerweise 1, aber wenn eines von ihnen 0 ist, weist das den Computer an, etwas Bestimmtes zu tun. Der Computer kann immer nur eine Sache auf einmal machen, so daß nicht mehr als eines dieser fünf Bits 0 sein sollte. Die Bits A5, A6, und A7 werden nicht beachtet, so daß Sie diese selbst nutzen können, wenn Sie ein Elektronikgenie sein sollten. Die brauchbarsten Adressen sind diejenigen, die 1 niedriger sind als ein Vielfaches von 32, demnach sind A0,..., A4 alle 1. Die Bits A8, A9 und so weiter werden manchmal dazu benützt, zusätzliche Informationen zu liefern.

Das gelesene oder geschriebene Byte hat 8 Bits. Sie werden oft (unter Verwendung von D für *Data*) bezeichnet als D7, D6,....., D1, D0. Hier eine Liste der verwendeten Bausteinadressen.

Es gibt einen Satz Eingabeadressen, die die Tastatur und auch die EAR-Buchse lesen.

Die Tastatur ist aufgeteilt in 8 Halbreihen von je 5 Tasten:

IN 65278 liest die Halbreihe **CAPS SHIFT** bis **V**
IN 65Ø22 liest die Halbreihe **A** bis **G**
IN 6451Ø liest die Halbreihe **Q** bis **T**
IN 63486 liest die Halbreihe **1** bis **5**
IN 61438 liest die Halbreihe **Ø** bis **6**
IN 57342 liest die Halbreihe **P** bis **Y**
IN 49150 liest die Halbreihe **ENTER** bis **H**
IN 32766 liest die Halbreihe **SPACE** bis **B**

(Diese Adressen sind 254+256*(255–2 ↑ n) beim Übergang *n* von Ø bis 7.)

Im eingelesenen Byte stehen die Bits DØ bis D4 für die fünf Tasten in der gegebenen Halbreihe – DØ für die äußerste Taste, D4 für die der Mitte am nächsten gelegene. Das Bit ist Ø, wenn die Taste gedrückt wird, und 1, wenn das nicht der Fall ist. D6 ist der Wert an der EAR-Buchse.

Die Bausteinadresse 254 in der Ausgabe steuert den Lautsprecher (D4) und die MIC-Buchse (D3) und setzt außerdem die Randfarbe (D2, D1 und DØ).

Die Bausteinadresse 251 steuert den Drucker beim Lesen und Schreiben. Das Lesen stellt fest, ob der Drucker wieder etwas übernehmen kann, das Schreiben sendet Punkte, die gedruckt werden.

Die Bausteinadressen 254, 247 und 239 werden für die in Kapitel 22 erwähnten Zusatzgeräte verwendet.

Fahren Sie dieses Programm:

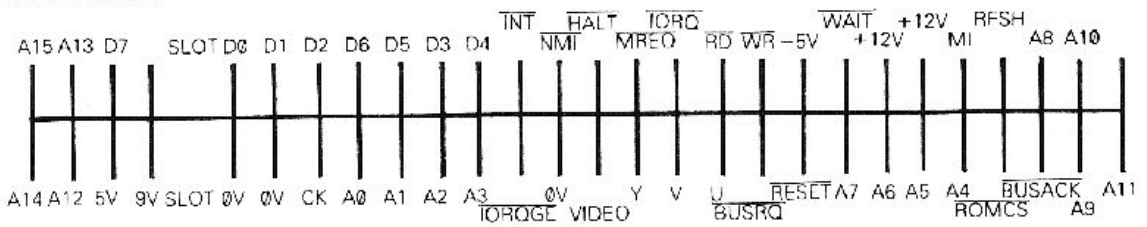
```
1Ø FOR n=Ø TO 7: REM Halbreihennummer
2Ø LET a=254+256*(255–2 ↑ n)
3Ø PRINT AT Ø,Ø; IN a: GO TO 3Ø
```

und drücken Sie beliebig Tasten. Wenn Sie von jeder Halbreihe genug haben, drücken Sie **BREAK** und geben ein

```
NEXT n
```

Die Steuer-, Date- und Adreßbuchsen liegen an der Rückseite des Spectrum offen, so daß Sie mit einem Spectrum nahezu alles machen können, wie mit einem Z 8Ø. Manchmal kann die Spectrum-Hardware aber im Weg sein. Hier eine Zeichnung der offenliegenden Anschlüsse an der Rückseite:

Bauteilseite



Unterseite

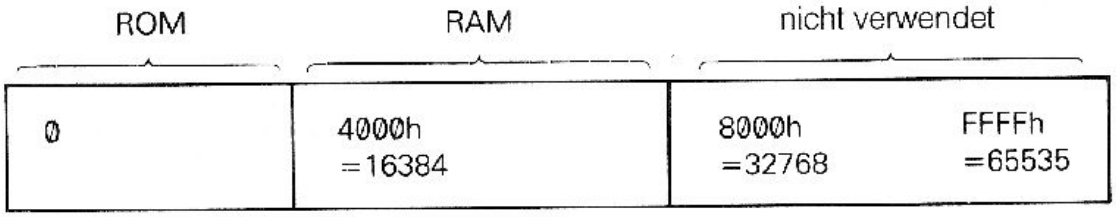
KAPITEL 24

Der Speicher

Zusammenfassung:
CLEAR

Ganz im Inneren des Computers wird alles gespeichert in Form von Bytes, das sind Zahlen zwischen Ø und 255. Sie mögen glauben, Sie hätten den Preis von Wolle oder die Adresse Ihres Düngemittellieferanten gespeichert, aber das ist alles umgewandelt worden in Ansammlungen von Bytes. Was der Computer sieht, sind Bytes.

Jeder Platz, wo ein Byte gespeichert werden kann, hat eine Adresse, das ist eine Zahl zwischen Ø und FFFFh (so daß eine Adresse als zwei Bytes gespeichert werden kann). Sie könnten sich den Speicher deshalb als eine lange Reihe von numerierten Fächern vorstellen, von denen jedes ein Byte aufnehmen kann. Allerdings sind nicht alle Fächer gleich. In der normalen 16K-RAM-Maschine fehlen die Fächer von 8ØØØh bis FFFFh völlig. Die Fächer von 4ØØØ bis 7FFFh sind RAM-Fächer, was bedeutet, daß Sie den Deckel aufmachen und den Inhalt verändern können; die von Ø bis 3FFFh sind ROM-Fächer, in die man hineinschauen kann wie durch einen Glasdeckel, die aber nicht zu öffnen sind. Sie müssen sich damit begnügen, das zu lesen, was hineingetan wurde, als man den Computer baute.



Um den Inhalt eines Faches zu besichtigen, verwenden wir die **PEEK**-Funktion. Ihr Argument ist die Adresse des Fachs, ihr Resultat der Inhalt. Beispiel: Das folgende Programm zeigt die ersten 21 Bytes im ROM (und ihre Adressen) an:

```
1Ø PRINT "Adresse"; TAB 1Ø; "Byte"
2Ø FOR a=Ø TO 2Ø
3Ø PRINT a; TAB 1Ø; PEEK a
4Ø NEXT a
```

Diese Bytes werden Ihnen natürlich alle nichts bedeuten, aber der Prozessorchip versteht sie als Anweisungen, die ihm sagen, was er tun soll.

Um den Inhalt eines Faches zu ändern (bei RAM), verwenden wir die **POKE**-Anweisung in der Form

POKE Adresse, neuer Inhalt

wobei 'Adresse' und 'neuer Inhalt' für numerische Ausdrücke stehen. Beispiel: Wenn Sie sagen

POKE 31ØØØ,57

erhält das Byte an Adresse 31000 den neuen Wert 57. Schreiben Sie

PRINT PEEK 31000

um sich das zeigen zu lassen. (Geben Sie andere Werte ein, um nachzuweisen, daß da nicht geschwindelt wird.) Der neue Wert muß zwischen –255 und +255 liegen. Ist er negativ, wird er um 256 erhöht.

Die Möglichkeit des Pokens verleiht Ihnen enorme Macht über den Computer, wenn Sie damit umgehen können, und gewaltige zerstörerische Kräfte, falls nicht. Durch Eingeben des falschen Werts in die falsche Adresse kann man sehr leicht Riesenprogramme verlieren, die einzutippen man Stunden aufgewendet hat. Zum Glück fügen Sie dem Computer damit keinen dauerhaften Schaden zu.

Wir wollen uns nun genauer ansehen, wie der RAM verwendet wird. Sie sollten das aber nur dann lesen, wenn es Sie wirklich interessiert.

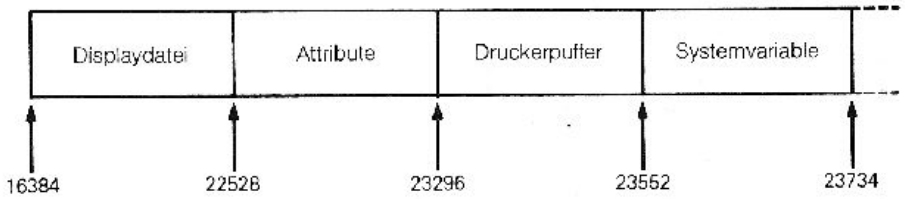
Der Speicher ist in verschiedene Bereiche aufgeteilt (erkennbar auf der großen Zeichnung), die verschiedene Arten von Informationen speichern. Die Bereiche reichen in der Größe für die Information, die sie enthalten, gerade aus, und wenn Sie an einem bestimmten Punkt mehr eingeben (etwa durch Zusatz einer Programmzeile oder einer Variablen), wird dadurch alles über diesen Punkt nach oben geschoben. Wenn Sie Informationen löschen, wird umgekehrt alles heruntergesetzt.

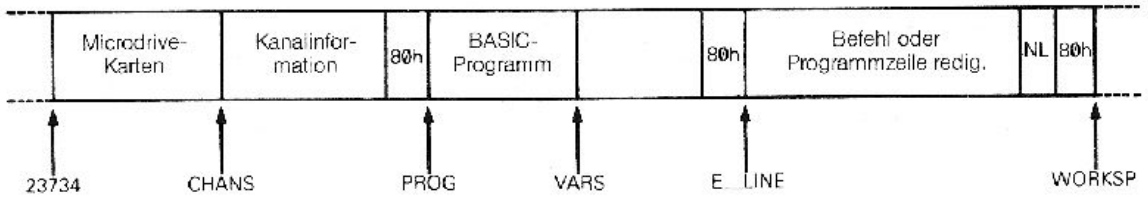
Die Displaydatei speichert das Fernsehbild. Sie ist eher seltsam angelegt, so daß es sich nicht so empfiehlt, mit **PEEK** oder **POKE** hineinzugehen. Jede Zeichenposition auf dem Bildschirm hat ein 8 mal 8 Punkte großes Quadrat, und jeder Punkt kann entweder 0 (Paper) oder 1 (Ink) sein. Durch die Verwendung von Binärzahlen können wir das Muster in Form von 8 Bytes speichern, für jede Reihe eines. Diese 8 Bytes werden aber nicht gemeinsam gespeichert. Die entsprechenden Reihen in den 32 Zeichen einer Einzelzeile werden gemeinsam als eine Abtastzeile von 32 Bytes gespeichert, weil der Elektronenstrahl im Fernseher das braucht, wenn er auf dem Bildschirm von links nach rechts abtastet. Da das vollständige Bild 24 Zeilen zu je 8 Abtastungen aufweist, könnte man annehmen, daß die ganzen 172 Abtastungen der Reihe nach hintereinander gespeichert werden; da würde man sich aber täuschen. Zuerst kommt die obere Abtastung der Zeilen 0 bis 7, dann die nächste Abtastung der Zeilen 0 bis 7, und so weiter bis zur untersten Abtastung der Zeilen 0 bis 7; anschließend dasselbe bei den Zeilen 8 bis 15 und erneut bei den Zeilen 16 bis 23. Die Folge des Ganzen: Wenn Sie an einen Computer gewöhnt sind, der für den Schirm **PEEK** und **POKE** verwendet, müssen Sie hier statt dessen **SCREEN$** und **PRINT AT** oder **PLOT** und **POINT** benützen.

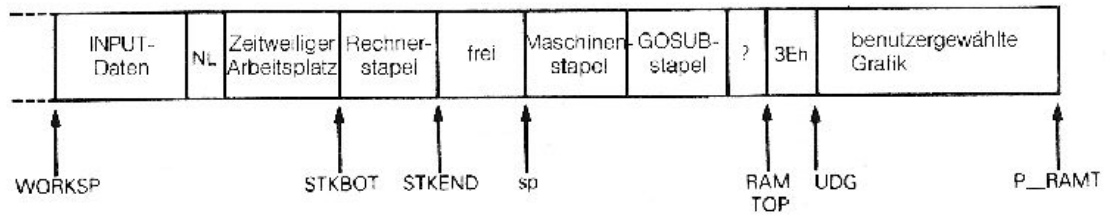
Die Attribute sind die Farben und dergleichen für jede Zeichenposition; verwendet wird das Format **ATTR**. Sie werden Zeile für Zeile in der Reihenfolge gespeichert, die man erwartet.

Der Druckerpuffer speichert die für den Drucker bestimmten Zeichen.

Die Systemvariablen enthalten verschiedene Informationen, die dem Computer mitteilen, in welcher Art von Zustand der Computer sich befindet. Im nächsten Kapitel werden sie vollständig aufgeführt, aber zunächst merken Sie sich einmal, daß es einige gibt (sie heißen CHANS, PROG, VARS, ELINE und so weiter), die die Adressen der Grenzen zwischen den verschiedenen Speicherabschnitten enthalten. Das sind keine BASIC-Variablen, ihre Namen erkennt der Computer nicht.


Displaydatei
Attribute
Druckerpuffer
Systemvariable
16384
22528
23296
23552
23734

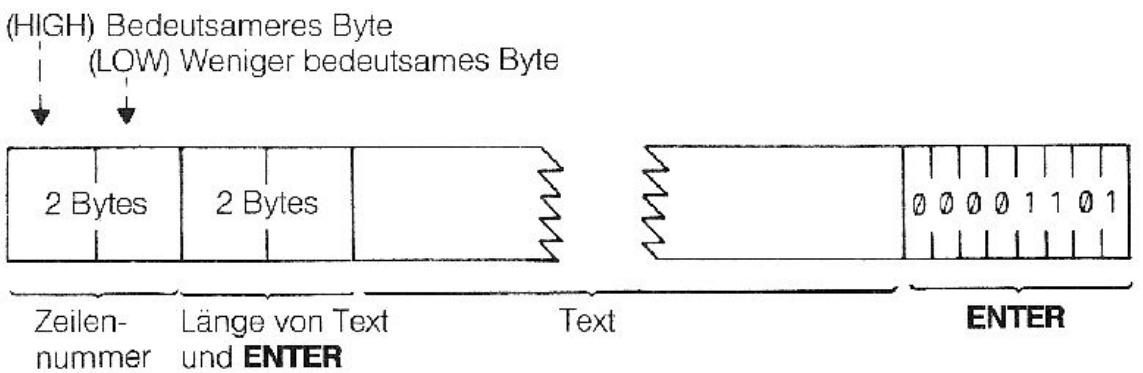

Microdrive-Karten
Kanalinformation
80h
BASIC-Programm
80h
Befehl oder Programmzeile redig.
NL
80h
23734
CHANS
PROG
VARS
E_LINE
WORKSP


INPUT-Daten
NL
Zeitweiliger Arbeitsplatz
Rechnerstapel
frei
Maschinenstapel
GOSUB-stapel
?
3Eh
benutzergewählte Grafik
WORKSP
STKBOT
STKEND
sp
RAM TOP
UDG
P_RAMT

Die Microdrive-Karten (also die Speicherplätze für das Laufwerk) werden nur zusammen mit diesem verwendet. Normalerweise befindet sich dort nichts.

Die Kanalinformation enthält Informationen über die Eingabe/Ausgabestellen, nämlich die Tastatur (mit der unteren Bildschirmhälfte), die obere Bildschirmhälfte und den Drucker.

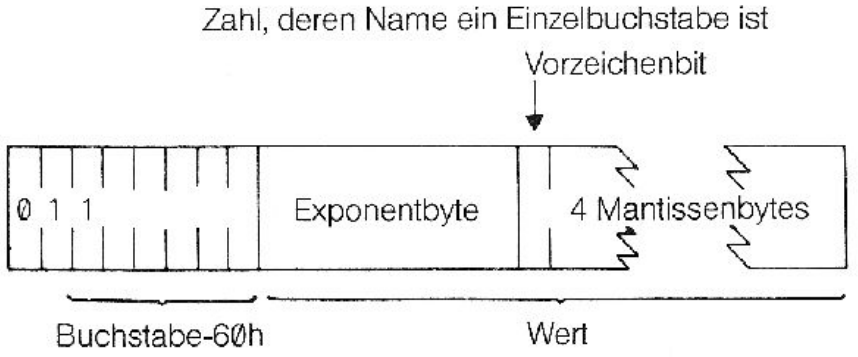
Jede Zeile BASIC-Programm hat die Form:

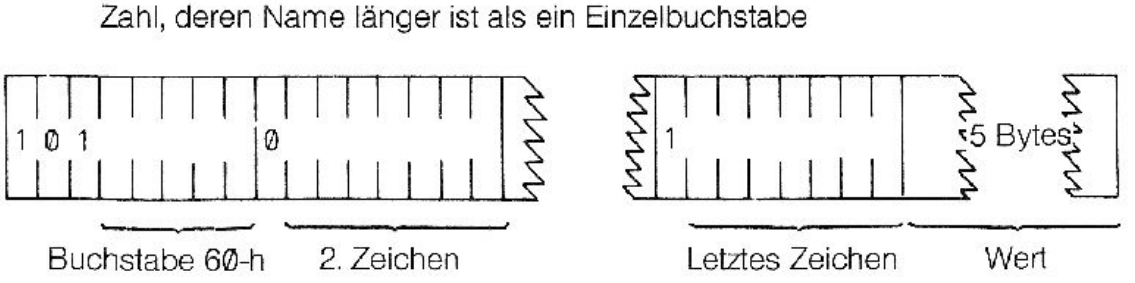


Beachten Sie, daß – im Gegensatz zu allen anderen Fällen von 2 Bytes langen Zahlen – im Z80-Mikroprozessor die Zeilennummer hier mit dem bedeutsameren Byte an erster Stelle gespeichert wird, das heißt, in der Reihenfolge, in der Sie sie niederschreiben.

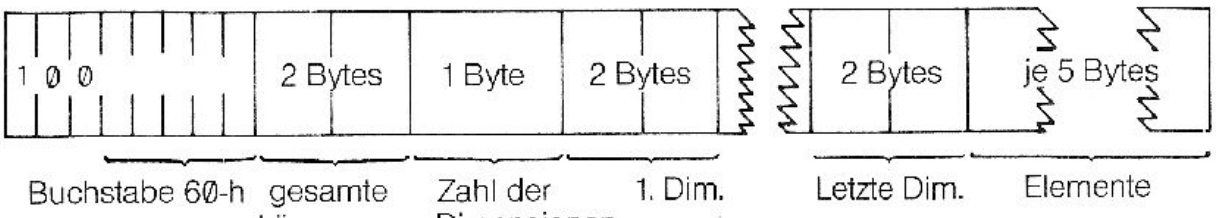
Einer numerischen Konstante im Programm folgt ihre binäre Form. Verwendet wird das Zeichen **CHR$ 14**, gefolgt von fünf Bytes für die Zahl selbst.

Die Variablen haben entsprechend ihrer unterschiedlichen Art verschiedene Formate. Die Buchstaben in den Namen hat man sich so vorzustellen, daß sie in Kleinschrift beginnen.





Zahlenarray:

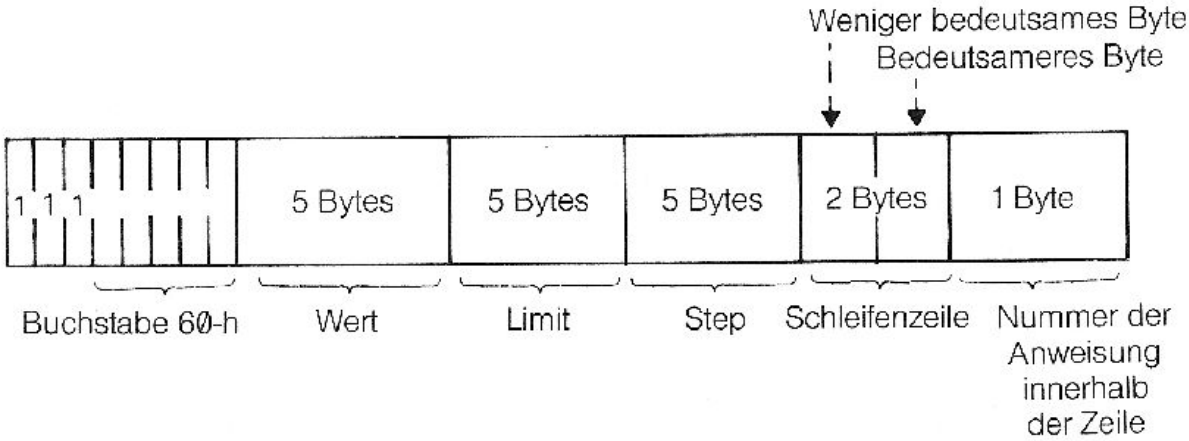


Die Reihenfolge der Elemente:
erstens die Elemente, für die der erste Index 1 ist,
dann die Elemente, für die der erste Index 2 ist,
dann die Elemente, für die der erste Index 3 ist,
und so weiter für alle möglichen Werte des ersten Index.

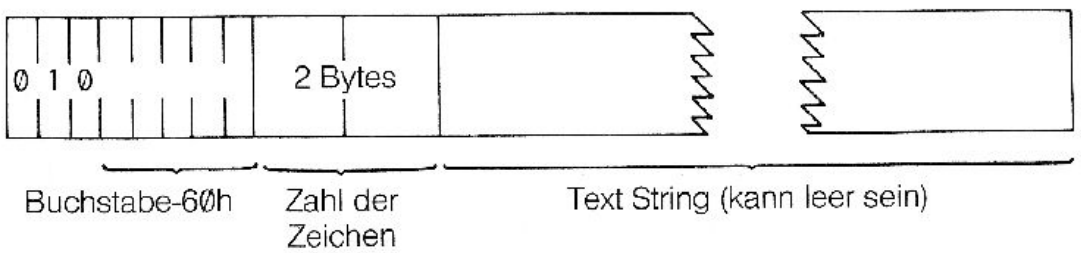
Die Elemente mit einem bestimmten ersten Index sind auf dieselbe Weise geordnet beim zweiten Index, und so hinunter bis zum letzten.

Beispiel: Die Elemente des 3*6-Arrays in Kapitel 12 werden gespeichert in der Reihenfolge b(1,1) b(1,2) b(1,3) b(1,4) b(1,5) b(1,6) b(2,1) b(2,2) ... b(2,6) n(3,1) b(3,2) ... b(3,6).

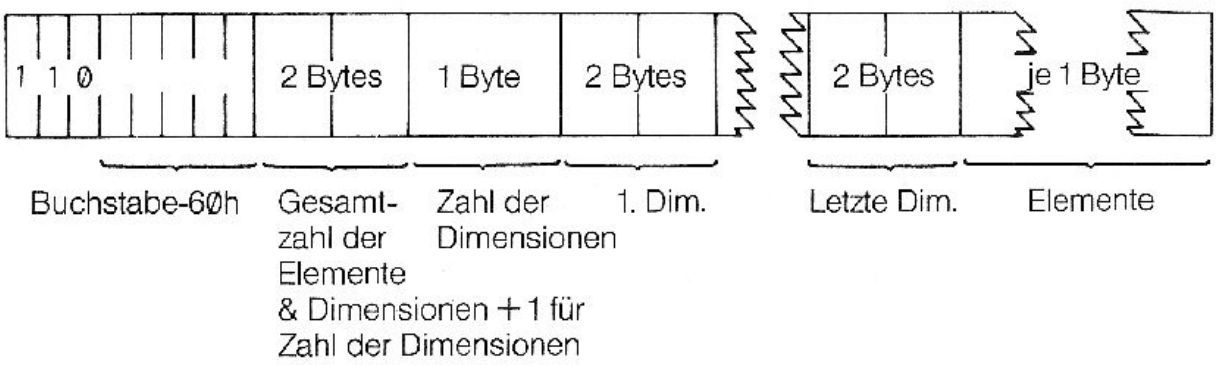
Steuervariable einer **FOR-NEXT**-Schleife:



String:



Zeichenarray:



Der Rechner ist Teil des BASIC-Systems, das mit Arithmetik zu tun hat. Die Zahlen, mit denen er umgeht, werden in erster Linie im Rechnerstapel aufbewahrt.

Der freie Platz enthält den bisher unbenutzten Raum.

Der Maschinenstapel ist jener, in dem der Z80-Prozessor Returnadressen und so weiter aufbewahrt.

Der **GOSUB**-Stapel wurde erwähnt in Kapitel 5.

Das Byte, auf das RAMTOP zeigt, besitzt die höchste Adresse im BASIC-System. Selbst **NEW**, das den RAM leert, geht nur bis hierhin, verändert also die benutzergewählten Grafikzeichen nicht. Sie können die Adresse RAMTOP verändern, wenn Sie eine Zahl in eine Löschanweisung setzen.

CLEAR neu RAMTOP

Das
(I) löscht alle Variablen
(II) löscht die Displaydatei (wie **CLS**)
(III) setzt die **PLOT**-Position auf die untere linke Ecke zurück
(IV) leistet **RESTORE**
(V) löscht den **GOSUB**-Stapel und setzt ihn auf das neue RAMTOP – vorausgesetzt, es liegt zwischen dem Rechnerstapel und dem materiellen Ende von RAM, im anderen Fall beläßt es RAMTOP, wie es war.

RUN bewirkt auch **CLEAR**, ohne aber je RAMTOP zu verändern.

Wenn Sie **CLEAR** auf diese Weise verwenden, können Sie RAMTOP entweder nach oben versetzen, um mehr Platz für BASIC zu schaffen, indem Sie die benutzergewählte Grafik überschreiben, oder nach unten setzen, um mehr RAM zu schaffen, was vor **NEW** geschützt ist.

Drücken Sie **NEW** und dann **CLEAR 23800**, um eine Vorstellung davon zu bekommen, was mit dem Gerät geschieht, wenn es voll wird.

Wenn Sie damit beginnen, ein Programm einzutippen, wird Ihnen mit als erstes auffallen, daß der Computer auf einmal nichts mehr annimmt und 'Tröööt' macht. Das heißt, der Computer ist rammelvoll, Sie müssen also etwas herausnehmen. Außerdem gibt es zwei Fehlermeldungen, die, grob gesprochen, einen ähnlichen Sinn haben: **4 Memory full** (Speicher voll) und **G No room for line** (Kein Platz für Zeile).

Der Summton ertönt auch, wenn Sie eine Programmzeile eingeben, die mehr als 23 Zeilen umfaßt. Dann wird Ihre Eingabe zwar nicht übersehen, obwohl Sie davon nichts merken, aber der Summton soll Sie davon abhalten, daß Sie weitermachen.

Die Länge des Summtons können Sie dadurch verändern, daß sie mit *POKE* eine Zahl in Adresse 23608 eingeben. Die übliche Länge hat die Zahl 64.

Jede Zahl (außer 0) kann ausschließlich geschrieben werden als

$$\pm m \times 2^e$$

wobei ± das Vorzeichen ist, m die *Mantisse*, zwischen ½ und 1 (1 kann sie nicht sein) und e der *Exponent*, eine ganze Zahl (möglicherweise negativ).

Angenommen, Sie schreiben m binär. Da es sich um einen Bruch handelt, wird er einen *Binärpunkt* besitzen (wie der Dezimalpunkt beim Zehnersystem – im Deutschen also ein Komma) und dann einen Binärbruch (wie ein Dezimalbruch): Binär wird also geschrieben:
ein Halbes .1
ein Viertel .01
drei Viertel .11
ein Zehntel .0001100110011001100110011... und so weiter. Bei unserer Zahl m gibt es, weil sie kleiner ist als 1, vor dem Binärpunkt keine Bits, und weil sie auf mindestens ½ steht, ist das Bit unmittelbar nach dem Binärpunkt eine 1.

Um die Zahl im Computer zu speichern, verwenden wir fünf Bytes, und zwar so:

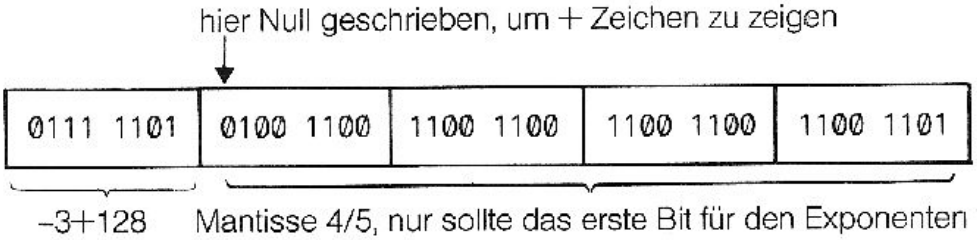
(I) Man schreibt die ersten acht Bits der Mantisse im zweiten Byte (daß das erste Bit 1 ist, wissen wir), die zweiten acht Bits im dritten Byte, die dritten acht Bits im vierten Byte und die vierten acht Bits im fünften Byte.

(II) Ersetzen Sie das erste Bit im zweiten Byte – von dem wir wissen, daß es 1 ist – durch das Vorzeichen: 0 für plus, 1 für minus,

(III) Schreiben Sie den Exponenten +128 im ersten Byte. Beispiel: Nehmen wir an, unsere Zahl sei 1/10

$$1/10 = 4/5 \times 2^{-3}$$

Dann ist die Mantisse m: .11001100110011001100110011001100 in binärer Schreibweise (weil das 33. Bit 1 ist, runden wir das 32. von 0 auf 1 auf), und der Exponent e^{-3}. Die Anwendung unserer drei Regeln liefert die fünf Bytes



Es gibt noch eine andere Art, ganze Zahlen zwischen –65535 und +65535 zu speichern:

(I) das erste Byte ist 0,

(II) das zweite Byte ist bei einer positiven Zahl 0, bei einer negativen FFh,

(III) die dritten und vierten Bytes sind die mehr und weniger bedeutsamen Bytes der Zahl (oder die Zahl +131072, wenn sie negativ ist),

(IV) das fünfte Byte ist 0.

KAPITEL 25

Die Systemvariablen

Die Speicherbytes von 23552 bis 23733 sind vom System für besondere Zwecke reserviert. Sie können mit *PEEK* hinein, um Verschiedenes über das System in Erfahrung zu bringen, manche kann man auch mit *POKE* sinnvoll verändern. Sie sind hier mit ihren Verwendungszwecken aufgeführt.

Sie werden *Systemvariable* genannt und besitzen Namen; Sie dürfen sie aber nicht mit den Variablen verwechseln, wie BASIC sie benützt. Der Computer erkennt die Namen als Hinweise auf Systemvariable nicht. Sie werden nur als Gedächtnishilfen für uns Menschenwesen mitgeteilt.

Die Abkürzungen in Spalte 1 haben folgende Bedeutung:

X Die Variable sollte nicht mit *POKE* verändert werden, weil sonst das System zusammenbrechen könnte,
N Hineingehen mit *POKE* hat keine Dauerwirkung.

Die Zahl in Spalte 1 ist die Zahl der Bytes in der Variablen. Bei zwei Bytes ist das erste das weniger bedeutsame Byte – im Gegensatz zu dem, was Sie erwartet haben mögen. Wenn Sie einer Variable mit 2 Bytes bei Adresse **n** also mit *POKE* einen Wert **v** geben wollen, verwenden Sie

```
POKE n,v-256*INT (v/256)
POKE n+1,INT (v/256)
```

und, um ihren Wert mit **PEEK** festzustellen, den Ausdruck

```
PEEK n+256*PEEK (n+1)
```

Hinweise	*Adresse*	*Name*	*Inhalt*
N8	23552	KSTATE	Verwendet beim Lesen der Tastatur.
N1	23560	LAST K	Speichert neu gedrückte Taste.
1	23561	REPDEL	Zeit (in 50tel Sekunden – Nordamerika 60tel Sek.), die eine Taste niedergedrückt sein muß, bevor sie wiederholt. Das beginnt bei 35, Sie können aber andere Werte mit **POKE** eingeben.
1	23562	REPPER	Verzögerung (in 50tel Sekunden – Nordamerika 60tel) zwischen aufeinanderfolgenden Wiederholungen einer gedrückten Taste: anfangs 5.
N2	23563	DEFADD	Adresse der Argumente von benutzergewählter Funktion, wenn eine behandelt wird; sonst 0.
N1	23565	K DATA	Speichert das 2. Byte der Farbsteuerungen, die über die Tastatur eingegeben werden.
N2	23566	TVDATA	Speichert Farbenbytes, **AT**- und **TAB**-Speicherungen, die zum Fernseher gehen.
X38	23568	STRMS	Adressen von Kanälen für Ströme

Hinweise	*Adresse*	*Name*	*Inhalt*
2	23606	CHARS	256 weniger als Adresse des Zeichenvorrats (der mit Leerstelle beginnt und bis zum Copyright-Symbol geht). Normal in ROM. Sie können aber Ihren eigenen in RAM setzen und CHARS darauf zeigen lassen.
1	23608	RASP	Länge des Warntons
1	23609	PIP	Länge des Tastaturklickens
1	23610	ERR NR	1 weniger als Meldecode. Beginnt bei 255 (für –1), also setzt **PEEK 23610** 255.
X1	23611	FLAGS	Verschiedene Flags zu Steuerung des BASIC-Systems.
X1	23612	TV FLAG	Flags im Zusammenhang mit TV-System.
X2	23613	ERR SP	Adresse von Posten auf Maschinenstapel, wird als Fehlerrücksprung verwendet.
N2	23615	LIST SP	Adresse der Rücksprungadresse von automatischem Listing.
N1	23617	MODE	Bestimmt K-, L-, C-, E- oder G-Cursor.
2	23618	NEWPPC	Zeile, zu der gesprungen werden soll.
1	23620	NSPPC	Anweisungsnummer in Zeile, zu der gesprungen werden soll. **POKE** zuerst bei NEWPPC und dann bei NSPPC erzwingt einen Sprung zu einer bestimmten Anweisung in einer Zeile.
2	23621	PPC	Zeilennummer der Anweisungszeile, die gerade ausgeführt wird.
1	23623	SUBPPC	Zahl innerhalb der Anweisungszeile, die ausgeführt wird.
1	23624	BORDCR	Randfarbe *8; enthält auch die Attribute, die normalerweise für die untere Bildschirmhälfte genutzt werden.
2	23625	E PPC	Nummer der laufenden Zeile (mit Programmcursor).
X2	23627	VARS	Adresse von Variablen.
N2	23629	DEST	Adresse von zugeteilter Variabler.
X2	23631	CHANS	Adresse von Kanaldaten.
X2	23633	CURCHL	Adresse von Information, die gerade für Ein- und Ausgabe verwendet wird.
X2	23635	PROG	Adresse von BASIC-Programm.
X2	23637	NXTLIN	Adresse der nächsten Zeile im Programm.
X2	23639	DATADD	Adresse für Begrenzer des letzten DATA-Postens.
X2	23641	E LINE	Adresse des Befehls, der eingetippt wird.
2	23643	K CUR	Adresse des Cursors.

Hinweise	*Adresse*	*Name*	*Inhalt*
X2	23645	CH ADD	Adresse des nächsten Zeichens, das interpretiert werden muß: das Zeichen nach dem Argument von **PEEK** oder das **NEWLINE** am Ende einer **POKE**-Anweisung.
2	23647	X PTR	Adresse des Zeichens nach dem ?-Kennzeichen.
X2	23649	WORKSP	Adresse des zeitweiligen Arbeitsplatzes.
X2	23651	STKBOT	Adresse Ende des Rechnerstapels.
X2	23653	STKEND	Adresse Anfang des freien Platzes.
N1	23655	BREG	Das b-Register des Rechners.
N2	23656	MEM	Adresse des Bereichs, der für Rechnerspeicher verwendet wird. (Gewöhnlich, aber nicht immer, MEMBOT.)
1	23658	FLAGS2	Weitere Flags.
X1	23659	DF SZ	Die Zahl der Zeilen (einschließlich einer Leerzeile) in der unteren Bildschirmhälfte.
2	23660	S TOP	Die Nummer der obersten Programmzeile bei automatischen Listings.
2	23662	OLDPPC	Zeilennummer, zu der **CONTINUE** springt.
1	23664	OSPCC	Zahl innerhalb Anweisungszeile, zu der **CONTINUE** springt.
N1	23665	FLAGX	Verschiedene Flags.
N2	23666	STRLEN	Länge des Ziels vom Stringtyp bei Zuteilung.
N2	23668	T ADDR	Adresse des nächsten Postens in Syntaxtabelle (kaum je sinnvoll zu gebrauchen).
2	23670	SEED	Der Keim für **RND**. Das ist die Variable, die von **RANDOMIZE** gesetzt wird.
3	23672	FRAMES	3 Byte-Blockzähler (das am wenigstens bedeutsame zuerst). Wird alle 20 ms erhöht. Siehe Kapitel 18.
2	23675	UDG	Adresse des ersten benutzergewählten Grafikzeichens. Sie können das verändern, z.B. um Platz zu sparen, indem Sie weniger benutzergewählte Grafikzeichen verwenden.
1	23677	COORDS	x-Koordinate des letzten Plotpunktes.
1	23678		y-Koordinate des letzten Plotpunktes.
1	23679	P POSN	33-Spaltennummer der Druckerposition.
1	23680	PR CC	Weniger bedeutsames Byte der Adresse für nächste Position, an der **LPRINT** drucken soll (im Druckerpuffer).
1	23681		Nicht verwendet.
2	23682	ECHO E	33-Spalten- und 24-Zeilen-Nummer vom Ende des Eingabepuffers (in der unteren Hälfte).
2	23684	DF CC	Adresse der **PRINT**-Position in Displaydatei.

Hinweise	*Adresse*	*Name*	*Inhalt*
2	23686	DFCCL	Wie DF CC für unteren Bildschirmteil.
X1	23688	S POSN	33-Spaltennummer für **PRINT**-Position.
X1	23689		24-Zeilen-Nummer für **PRINT**-Position.
X2	23690	SPOSNL	Wie S POSN für unteren Teil.
1	23692	SCR CT	Zählt Abrollungen. Hat immer 1 mehr als die Zahl der Abrollungen, die stattfindet, bevor mit **scroll?** angehalten wird. Wenn Sie mit **POKE** eine höhere Zahl als 1 eingeben (etwa 255), rollt der Schirm ständig ab, ohne Sie zu fragen.
1	23693	ATTR P	Ständige laufende Farben etc. (wie von Farbanweisungen eingegeben).
1	23694	MASK P	Verwendet für durchscheinende Farben etc. Jedes Bit, das 1 ist, zeigt, daß das dazugehörige Attribut-Bit nicht von ATTR P genommen wird, sondern von dem, was bereits auf dem Bildschirm steht.
N1	23695	ATTR T	Zeitweilige laufende Farben etc. (wie von Farbenposten eingegeben).
N1	23696	MASK T	Wie MASK P, aber vorübergehend.
1	23697	P FLAG	Weitere Flags.
N30	23698	MEMBOT	Speicherbereich des Rechners; wird dazu verwendet, Zahlen zu speichern, die gerade nicht auf den Rechnerspeicher gesetzt werden können.
2	23728		Nicht verwendet.
2	23730	RAMTOP	Adresse des letzten Byte vom Bereich des BASIC-Systems.
2	23732	P-RAMT	Adresse des letzten Byte des materiellen RAM.

Dieses Programm liefert Ihnen die ersten 22 Bytes des Variablenbereichs:

```
10 FOR n=0 TO 21
20 PRINT PEEK (PEEK 23627+256*PEEK 23628+n)
30 NEXT n
```

Versuchen Sie, die Übereinstimmung der Steuervariable **n** mit den obigen Beschreibungen zu finden.

Verändern Sie dann Zeile 20 zu

```
20 PRINT PEEK (23755+n)
```

Das teilt Ihnen die ersten 22 Bytes des Programmbereichs mit. Bringen Sie sie in Verbindung mit dem Programm.

KAPITEL 26

Umgang mit Maschinencode

Zusammenfassung:
USR mit numerischem Argument

Dieses Kapitel ist für diejenigen geschrieben, die *Z80-Maschinencode* verstehen, den Befehlssatz, den der Z80-Mikroprozessor verwendet. Wenn das bei Ihnen nicht der Fall ist, Sie es aber lernen möchten: Es gibt viele Bücher darüber. Sie sollten sich eines mit dem ungefähren Titel 'Z80-Maschinencode (oder Assemblersprache) für den absoluten Anfänger' besorgen. Wenn der Spectrum erwähnt wird, ist das natürlich noch besser.

Diese Programme sind gewöhnlich in Assemblersprache geschrieben. Sie erscheinen auf den ersten Blick zwar ein wenig rätselhaft, sind aber mit etwas Übung nicht schwer zu verstehen. (Die Anweisungen in Assemblersprache sehen Sie in Anhang A.) Will man sie aber mit dem Computer fahren, muß man das Programm in einer Folge von Bytes codieren – in dieser Form wird sie *Maschinencode* genannt. Die Übersetzung erfolgt in der Regel durch den Computer selbst, und zwar mit Hilfe eines Programms, das *Assembler* genannt wird. Im Spectrum ist kein Assembler eingebaut, aber Sie können in vielen Fällen einen auf Kassette kaufen. Wenn das nicht möglich sein sollte, müssen Sie die Übersetzung selbst vornehmen, vorausgesetzt, das Programm ist nicht übermäßig lang.

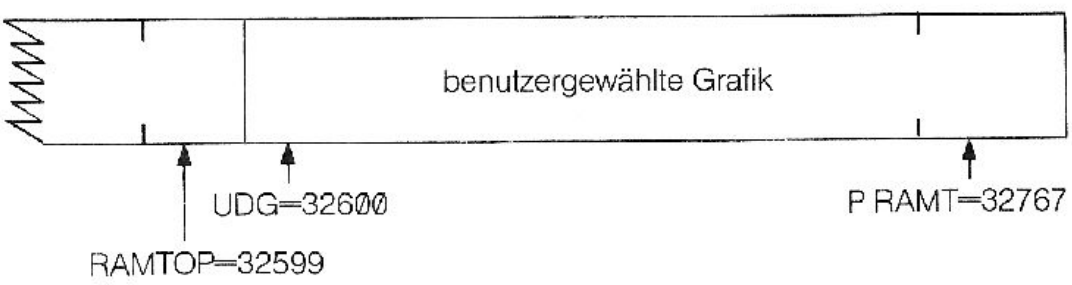
Nehmen wir als Beispiel das Programm

```
ld bc, 99
ret
```

Das Registerpaar bc wird dadurch mit 99 geladen. Das wird übersetzt in die vier Maschinencodebytes 1, 99, 0 (für ld bc, 99) und 201 (für ret). (Wenn Sie in Anhang A1 und 201 nachschlagen, finden Sie ld bc, NN – wobei NN für jede Zahl von zwei Bytes steht – und ret.)

Wenn Sie Ihr Maschinencode-Programm haben, ist der nächste Schritt, es in den Computer einzugeben. (Ein Assembler würde das vermutlich automatisch machen.) Sie müssen entscheiden, wo im Speicher es hin soll. Am besten macht man zwischen dem BASIC-Bereich und den benutzergewählten Grafikzeichen eigens Platz dafür.

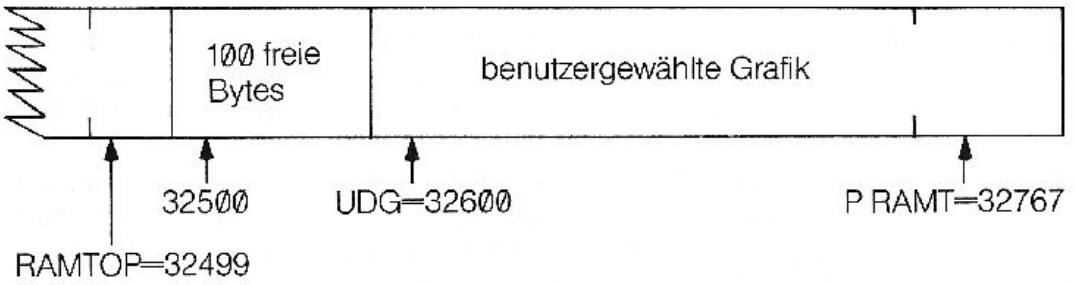
Nehmen wir, zum Beispiel, an, Sie haben einen 16K-Spectrum. Das obere Ende von RAM hat da



Wenn Sie schreiben

CLEAR 32499

erhalten Sie einen Raum von 100 Bytes (großzügig bemessen), beginnend bei Adresse 32500.



Um das Maschinencode-Programm einzugeben, würden Sie ein BASIC-Programm verwenden in der Art von

```
10 LET a=32500
20 READ n: POKE a,n
30 LET a=a+1: GO TO 20
40 DATA 1,99,0,201
```

(Das hält an mit der Meldung **E Out of DATA**, wenn die vier angegebenen Bytes eingefügt sind.)

Um den Maschinencode zu fahren, verwenden Sie die Funktion **USR**, diesmal aber mit einem numerischen Argument, der Startadresse. Ihr Resultat ist der Wert des bc-Registers bei der Rückkehr aus dem Maschinencode-Programm. Wenn Sie also schreiben

PRINT USR 32500

erhalten Sie die Antwort 99.

Die Rücksprungadresse zu BASIC ist in der üblichen Weise gestapelt, so daß die Rückkehr über eine Z80-ret-Anweisung erfolgt. Sie sollten bei einer Maschinencode-Routine die Register iy und i nicht verwenden.

Ihr Maschinencode-Programm können Sie mühelos sichern mit

SAVE "irgendein Name" **CODE 32500,4**

Auf Anhieb gibt es keinen Weg, das so zu sichern, daß es automatisch selbst läuft, wenn es geladen wird, aber das können Sie mit Hilfe eines BASIC-Programms umgehen.

```
10 LOAD "" CODE 32500,4
20 PRINT USR 32500
```

Machen Sie zuerst

SAVE "irgendein Name" **LINE**

und dann

SAVE "xxxx" CODE 325ØØ,4
LOAD "irgendein Name"

Das BASIC-Programm wird dann geladen und automatisch gefahren, und es lädt und fährt den Maschinencode.

Der Zeichenvorrat

Das ist der vollständige Spectrum-Zeichenvorrat mit den Codes in Dezimal und Hexadezimal. Wenn man sich die Codes als Z80-Maschinencode-Anweisungen vorstellt, liefern die rechten Spalten die entsprechenden mnemotechnischen Notationen in Assemblersprache. Wie Sie wahrscheinlich wissen, wenn Sie sich mit diesen Dingen auskennen, sind bestimmte Z80-Befehle zusammengesetzte Ausdrücke, die mit CBh oder EDh beginnen; die beiden Spalten rechts außen geben sie an.

Code	*Zeichen*	*Hex*	*Z80-Assembler*	*– nach CB*	*– nach ED*
0	nicht verwendet	00	nop	rlc b	
1		01	ld bc,NN	rlc c	
2		02	ld (bc),a	rlc d	
3		03	inc bc	rlc e	
4		04	inc b	rlc h	
5		05	dec b	rlc l	
6	**PRINT** Komma	06	ld b,N	rlc (hl)	
7	**EDIT**	07	rlca	rlc a	
8	Cursor links	08	ex af,af'	rrc b	
9	Cursor rechts	09	add hl,bc	rrc c	
10	Cursor abwärts	0A	ld a,(bc)	rrc d	
11	Cursor aufwärts	0B	dec bc	rrc e	
12	**DELETE**	0C	inc c	rrc h	
13	**ENTER**	0D	dec c	rrc l	
14	Zahl	0E	ld c,N	rrc (hl)	
15	nicht verwendet	0F	rrca	rrc a	
16	**INK** Steuerung	10	djnz DIS	rl b	
17	**PAPER** Steuerung	11	ld de,NN	rl c	
18	**FLASH** Steuerung	12	ld (de),a	rl d	
19	**BRIGHT** Steuerung	13	inc de	rl e	
20	**INVERSE** Steuerung	14	inc d	rl h	
21	**OVER** Steuerung	15	dec d	rl l	
22	**AT Steuerung**	16	ld d,N	rl (hl)	
23	**TAB** Steuerung	17	rla	rl a	
24	nicht verwendet	18	jr DIS	rr b	
25		19	add hl,de	rr c	
26		1A	ld a,(de)	rr d	
27		1B	dec de	rr e	
28		1C	inc e	rr h	
29		1D	dec e	rr l	
30		1E	ld e,N	rr (hl)	
31		1F	rra	rr a	
32	Leerraum	20	jr nz,DIS	sla b	
33	!	21	ld hl,NN	sla c	
34	"	22	ld (NN),hl	sla d	
35	#	23	inc hl	sla e	
36	$	24	inc h	sla h	
37	%	25	dec h	sla l	

Code	*Zeichen*	*Hex*	*Z80 Assembler*	*– nach CB*	*– nach ED*
38	&	26	ld h,N	sla (hl)	
39	'	27	daa	sla a	
40	(	28	jr z,DIS	sra b	
41	)	29	add hl,hl	sra c	
42	*	2A	ld hl,(NN)	sra d	
43	+	2B	dec hl	sra e	
44	,	2C	inc l	sra h	
45	–	2D	dec l	sra l	
46	.	2E	ld l,N	sra (hl)	
47	/	2F	cpl	sra a	
48	0	30	jr nc,DIS		
49	1	31	ld sp,NN		
50	2	32	ld (NN),a		
51	3	33	inc sp		
52	4	34	inc (hl)		
53	5	35	dec (hl)		
54	6	36	ld (hl),N		
55	7	37	scf		
56	8	38	jr c,DIS	srl b	
57	9	39	add hl,sp	srl c	
58	:	3A	ld a,(NN)	srl d	
59	;	3B	dec sp	srl e	
60	<	3C	inc a	srl h	
61	=	3D	dec a	srl l	
62	>	3E	ld a,N	srl (hl)	
63	?	3F	ccf	srl a	
64	@	40	ld b,b	bit 0,b	in b,(c)
65	A	41	ld b,c	bit 0,c	out (c),b
66	B	42	ld b,d	bit 0,d	sbc hl,bc
67	C	43	ld b,e	bit 0,e	ld (NN),bc
68	D	44	ld b,h	bit 0,h	neg
69	E	45	ld b,l	bit 0,l	retn
70	F	46	ld b,(hl)	bit 0,(hl)	im 0
71	G	47	ld b,a	bit 0,a	ld i,a
72	H	48	ld c,b	bit 1,b	in c,(c)
73	I	49	ld c,c	bit 1,c	out (c),c
74	J	4A	ld c,d	bit 1,d	adc hl,bc
75	K	4B	ld c,e	bit 1,e	ld bc,(NN)
76	L	4C	ld c,h	bit 1,h	
77	M	4D	ld c,l	bit 1,l	reti
78	N	4E	ld c,(hl)	bit 1,(hl)	
79	O	4F	ld c,a	bit 1,a	ld r,a
80	P	50	ld d,b	bit 2,b	in d,(c)
81	Q	51	ld d,c	bit 2,c	out (c),d

Code	*Zeichen*	*Hex*	*Z80 Assembler*	*– nach CB*	*– nach ED*
82	R	52	ld d,d	bit 2,d	sbc hl,de
83	S	53	ld d,e	bit 2,e	ld (NN),de
84	T	54	ld d,h	bit 2,h	
85	U	55	ld d,l	bit 2,l	
86	V	56	ld d,(hl)	bit 2,(hl)	im 1
87	W	57	ld d,a	bit 2,a	ld a,i
88	X	58	ld e,b	bit 3,b	in e,(c)
89	Y	59	ld e,c	bit 3,c	out (c),e
90	Z	5A	ld e,d	bit 3,d	adc hl,de
91	[	5B	ld e,e	bit 3,e	ld de,(NN)
92	/	5C	ld e,h	bit 3,h	
93	]	5D	ld e,l	bit 3,l	
94	↑	5E	ld e,(hl)	bit 3,(hl)	im 2
95		5F	ld e,a	bit 3,a	ld a,r
96	£	60	ld h,b	bit 4,b	in h,(c)
97	a	61	ld h,c	bit 4,c	out (c),h
98	b	62	ld h,d	bit 4,d	sbc hl,hl
99	c	63	ld h,e	bit 4,e	ld (NN),hl
100	d	64	ld h,h	bit 4,h	
101	e	65	ld h,l	bit 4,l	
102	f	66	ld h,(hl)	bit 4,(hl)	
103	g	67	ld h,a	bit 4,a	rrd
104	h	68	ld l,b	bit 5,b	in l,(c)
105	i	69	ld l,c	bit 5,c	out (c),l
106	j	6A	ld l,d	bit 5,d	adc hl,hl
107	k	6B	ld l,e	bit 5,e	ld hl,(NN)
108	l	6C	ld l,h	bit 5,h	
109	m	6D	ld l,l	bit 5,l	
110	n	6E	ld l,(hl)	bit 5,(hl)	
111	o	6F	ld l,a	bit 5,a	rld
112	p	70	ld (hl),b	bit 6,b	in f,(c)
113	q	71	ld (hl),c	bit 6,c	
114	r	72	ld (hl),d	bit 6,d	sbc hl,sp
115	s	73	ld (hl),e	bit 6,e	ld (NN),sp
116	t	74	ld (hl),h	bit 6,h	
117	u	75	ld (hl),l	bit 6,l	
118	v	76	halt	bit 6,(hl)	
119	w	77	ld (hl),a	bit 6,a	
120	x	78	ld a,b	bit 7,b	in a,(c)
121	y	79	ld a,c	bit 7,c	out (c),a
122	z	7A	ld a,d	bit 7,d	adc hl,sp
123	{	7B	ld a,e	bit 7,e	ld sp,(NN)
124	|	7C	ld a,h	bit 7,h	
125	}	7D	ld a,l	bit 7,l	

Code	*Zeichen*	*Hex*	*Z80 Assembler*	*– nach CB*	*– nach ED*
126	-	7E	ld a,(hl)	bit 7,(hl)	
127	©	7F	ld a,a	bit 7,a	
128		80	add a,b	res 0,b	
129		81	add a,c	res 0,c	
130		82	add a,d	res 0,d	
131		83	add a,e	res 0,e	
132		84	add a,h	res 0,h	
133		85	add a,l	res 0,l	
134		86	add a,(hl)	res 0,(hl)	
135		87	add a,a	res 0,a	
136		88	adc a,b	res 1,b	
137		89	adc a,c	res 1,c	
138		8A	adc a,d	res 1,d	
139		8B	adc a,e	res 1,e	
140		8C	adc a,h	res 1,h	
141		8D	adc a,l	res 1,l	
142		8E	adc a,(hl)	res 1,(hl)	
143		8F	adc a,a	res 1,a	
144	(a)	90	sub b	res 2,b	
145	(b)	91	sub c	res 2,c	
146	(c)	92	sub d	res 2,d	
147	(d)	93	sub e	res 2,e	
148	(e)	94	sub h	res 2,h	
149	(f)	95	sub l	res 2,l	
150	(g)	96	sub (hl)	res 2,(hl)	
151	(h)	97	sub a	res 2,a	
152	(i)	98	sbc a,b	res 3,b	
153	(j)	99	sbc a,c	res 3,c	
154	(k)	9A	sbc a,d	res 3,d	
155	(l)	9B	sbc a,e	res 3,e	
156	(m)	9C	sbc a,h	res 3,h	
157	(n)	9D	sbc a,l	res 3,l	
158	(o)	9E	sbc a,(hl)	res 3,(hl)	
159	(p)	9F	sbc a,a	res 3,a	
160	(q)	A0	and b	res 4,b	ldi
161	(r)	A1	and c	res 4,c	cpi
162	(s)	A2	and d	res 4,d	ini
163	(t)	A3	and e	res 4,e	outi
164	(u)	A4	and h	res 4,h	
165	**RND**	A5	and l	res 4,l	
166	**INKEY$**	A6	and (hl)	res 4,(hl)	
167	**PI**	A7	and a	res 4,a	
168	**FN**	A8	xor b	res 5,b	ldd
169	**POINT**	A9	xor c	res 5,c	cpd

(144–164: user graphics)

Code	*Zeichen*	*Hex*	*Z80 Assembler*	*– nach CB*	*– nach ED*
170	**SCREEN$**	AA	xor d	res 5,d	ind
171	**ATTR**	AB	xor e	res 5,e	outd
172	**AT**	AC	xor h	res 5,h	
173	**TAB**	AD	xor l	res 5,l	
174	**VAL$**	AE	xor (hl)	res 5,(hl)	
175	**CODE**	AF	xor a	res 5,a	
176	**VAL**	B0	or b	res 6,b	ldir
177	**LEN**	B1	or c	res 6,c	cpir
178	**SIN**	B2	or d	res 6,d	inir
179	**COS**	B3	or e	res 6,e	otir
180	**TAN**	B4	or h	res 6,h	
181	**ASN**	B5	or l	res 6,l	
182	**ACS**	B6	or (hl)	res 6,(hl)	
183	**ATN**	B7	or a	res 6,a	
184	**LN**	B8	cp b	res 7,b	lddr
185	**EXP**	B9	cp c	res 7,c	cpdr
186	**INT**	BA	cp d	res 7,d	indr
187	**SQR**	BB	cp e	res 7,e	otdr
188	**SGN**	BC	cp h	res 7,h	
189	**ABS**	BD	cp l	res 7,l	
190	**PEEK**	BE	cp (hl)	res 7,(hl)	
191	**IN**	BF	cp a	res 7,a	
192	**USR**	C0	ret nz	set 0,b	
193	**STR$**	C1	pop bc	set 0,c	
194	**CHR$**	C2	jp nz,NN	set 0,d	
195	**NOT**	C3	jp NN	set 0,e	
196	**BIN**	C4	call nz,NN	set 0,h	
197	**OR**	C5	push bc	set 0,l	
198	**AND**	C6	add a,N	set 0,(hl)	
199	<=	C7	rst 0	set 0,a	
200	>=	C8	ret z	set 1,b	
201	<>	C9	ret	set 1,c	
202	**LINE**	CA	jp z,NN	set 1,d	
203	**THEN**	CB		set 1,e	
204	**TO**	CC	call z,NN	set 1,h	
205	**STEP**	CD	call NN	set 1,l	
206	**DEF FN**	CE	adc a,N	set 1,(hl)	
207	**CAT**	CF	rst 8	set 1,a	
208	**FORMAT**	D0	ret nc	set 2,b	
209	**MOVE**	D1	pop de	set 2,c	
210	**ERASE**	D2	jp nc,NN	set 2,d	
211	**OPEN #**	D3	out (N),a	set 2,e	
212	**CLOSE #**	D4	call nc,NN	set 2,h	
213	**MERGE**	D5	push de	set 2,l	

Code	*Charakter*	*Hex*	*Z80 Assembler*	*– nach CB*	*– nach ED*
214	**VERIFY**	D6	sub N	set 2,(hl)	
215	**BEEP**	D7	rst 16	set 2,a	
216	**CIRCLE**	D8	ret c	set 3,b	
217	**INK**	D9	exx	set 3,c	
218	**PAPER**	DA	jp c,NN	set 3,d	
219	**FLASH**	DB	in a,(N)	set 3,e	
220	**BRIGHT**	DC	call c,NN	set 3,h	
221	**INVERSE**	DD	prefixes instructions using ix	set 3,l	
222	**OVER**	DE	sbc a,N	set 3,(hl)	
223	**OUT**	DF	rst 24	set 3,a	
224	**LPRINT**	E0	ret po	set 4,b	
225	**LLIST**	E1	pop hl	set 4,c	
226	**STOP**	E2	jp po,NN	set 4,d	
227	**READ**	E3	ex (sp),hl	set 4,e	
228	**DATA**	E4	call po,NN	set 4,h	
229	**RESTORE**	E5	push hl	set 4,l	
230	**NEW**	E6	and N	set 4,(hl)	
231	**BORDER**	E7	rst 32	set 4,a	
232	**CONTINUE**	E8	ret pe	set 5,b	
233	**DIM**	E9	jp (hl)	set 5,c	
234	**REM**	EA	jp pe,NN	set 5,d	
235	**FOR**	EB	ex de,hl	set 5,e	
236	**GO TO**	EC	call pe,NN	set 5,h	
237	**GO SUB**	ED		set 5,l	
238	**INPUT**	EE	xor N	set 5,(hl)	
239	**LOAD**	EF	rst 40	set 5,a	
240	**LIST**	F0	ret p	set 6,b	
241	**LET**	F1	pop af	set 6,c	
242	**PAUSE**	F2	jp p,NN	set 6,d	
243	**NEXT**	F3	di	set 6,e	
244	**POKE**	F4	call p,NN	set 6,h	
245	**PRINT**	F5	push af	set 6,l	
246	**PLOT**	F6	or N	set 6,(hl)	
247	**RUN**	F7	rst 48	set 6,a	
248	**SAVE**	F8	ret m	set 7,b	
249	**RANDOMIZE**	F9	ld sp,hl	set 7,c	
250	**IF**	FA	jp m,NN	set 7,d	
251	**CLS**	FB	ei	set 7,e	
252	**DRAW**	FC	call m,NN	set 7,h	
253	**CLEAR**	FD	prefixes instructions using iy	set 7,l	
254	**RETURN**	FE	cp N	set 7,(hl)	
255	**COPY**	FF	rst 56	set 7,a	

Meldungen

Sie erscheinen unten am Bildschirm, sobald der Computer damit aufhört, BASIC-Anweisungen auszuführen. Sie erklären, warum er aufgehört hat, ob aus einem natürlichen Grund, oder weil ein Fehler aufgetreten ist.

Die Meldung besteht aus einer Codezahl oder einem Codebuchstaben, die Sie hier in dieser Tabelle nachschlagen können, einer kurzen Mitteilung, die erklärt, was geschehen ist, und der Zeilennummer und Anweisungsnummer innerhalb dieser Zeile, wo der Computer aufgehört hat. (Ein Befehl wird angezeigt als Zeile Ø. Innerhalb einer Zeile steht Anweisung 1 am Anfang, Anweisung 2 kommt nach dem ersten Doppelpunkt oder **THEN**, und so weiter.)

Wie **CONTINUE** sich verhält, hängt sehr von der Art der Meldung ab. Normalerweise geht **CONTINUE** zu der Zeile und Anweisung, die in der letzten Meldung angegeben wurde, aber es gibt Ausnahmen bei den Meldungen Ø, 9 und D (siehe auch Anhang C).

Hier eine Tabelle mit sämtlichen Meldungen. Sie teilt auch mit, unter welchen Umständen die Meldung auftreten kann, und das verweist Sie auf Anhang C. Beispiel: Fehlermeldung **A Invalid argument** (nicht zulässiges Argument) kann vorkommen bei **SQR, IN, ACS** und **ASN**, und die Einträge für sie in Anhang C erklären Ihnen genau, welche Argumente unzulässig sind.

Code	*Bedeutung*	*Situationen*
Ø	OK Erfolgreicher Abschluß oder Sprung zu einer höheren Zeilennummer als vorhanden. Die Meldung verändert die Zeile und Anweisung nicht, wohin **CONTINUE** springt.	Jede
1	NEXT without FOR Die Steuervariable existiert nicht (sie ist nicht durch eine **FOR**-Anweisung gesetzt worden), aber es gibt eine gewöhnliche Variable mit demselben Namen.	**NEXT**
2	Variable not found Bei einer einfachen Variablen geschieht das, wenn die Variable verwendet wird, bevor sie in einer **LET-, READ-** oder **INPUT**-Anweisung zugeteilt oder vom Band geladen oder in einer **FOR**-Anweisung gesetzt worden ist. Bei einer indizierten Variablen tritt es ein, wenn die Variable verwendet wird, bevor sie in einer **DIM**-Anweisung dimensioniert oder vom Band geladen wurde.	Jede
3	Subscript wrong Ein Index liegt außerhalb der Dimension des Arrays oder die Zahl der Indices ist falsch. Ist der Index negativ oder größer als 65535, entsteht Fehlermeldung B.	Indizierte Variable, Substrings

Code	*Bedeutung*	*Situation*
4	Out of memory Für das, was Sie tun wollen, ist im Computer nicht genug Platz. Wenn der Computer in diesem Zustand wirklich steckenzubleiben scheint, müssen Sie vielleicht mit **DELETE** die Kommandozeile beseitigen und dann ein, zwei Programmzeilen löschen (in der Absicht, sie später wieder einzufügen), damit Sie sich Spielraum verschaffen, am besten mit **CLEAR**.	**LET, INPUT, FOR, DIM, GO SUB, LOAD, MERGE**. Manchmal bei Behandlung von Ausdrücken.
5	Out of Screen Eine **INPUT**-Anweisung hat versucht, in der unteren Bildschirmhäfte mehr als 23 Zeilen zu erzeugen. Kommt auch vor bei **PRINT AT** 22, ...	**INPUT, PRINT AT**
6	Number too big Die Berechnungen haben zu einer Zahl geführt, die größer ist als rund $1Ø^{38}$.	Jede Art Arithmetik
7	RETURN without GO SUB Es hat ein **RETURN** mehr gegeben als **GO SUB**-Anweisungen.	**RETURN**
8	End of file	Arbeiten mit Microdrive, etc.
9	STOP statement Danach wiederholt **CONTINUE** das **STOP** nicht, sondern fährt mit der nächsten Anweisung fort.	**STOP**
A	Invalid argument Das Argument für eine Funktion ist aus irgendeinem Grund nicht richtig.	**SQR, LN, ASN, ACS, USR** (mit Stringargument)
B	Integer out of range Wenn eine ganze Zahl (Integer) verlangt ist, wird das Fließpunktargument zur nächsten ganzen Zahl abgerundet. Fehler B tritt auf, wenn sie außerhalb eines angemessenen Bereichs liegt. Für Arrayzugriff siehe auch Fehler 3.	**RUN, RANDOMIZE, POKE, DIM, GO TO, GO SUB, LIST, LLIST, PAUSE, PLOT, CHR$, PEEK, USR** (mit numerischem Argument) Arrayzugriff
C	Nonsense in BASIC Der Text des Stringarguments stellt keinen zulässigen Ausdruck dar.	**VAL, VAL$**

Code	*Bedeutung*	*Situation*
D	BREAK – CONT repeats Während einer peripheren Operation wurde **BREAK** gedrückt. Das Verhalten von **CONTINUE** nach dieser Meldung ist insoweit normal, daß es die Anweisung wiederholt. Vergleiche mit der Meldung L.	**LOAD, SAVE, VERIFY, MERGE, LPRINT, LLIST, COPY**. Auch, wenn der Computer **scroll?** fragt und Sie **N** oder **SPACE** oder **STOP** geben.
E	OUT of DATA Sie haben versucht, über das Ende der **DATA**-Liste hinauszulesen.	**READ**
F	Invalid file name **SAVE** mit einem Namen, der leer oder länger ist als 10 Zeichen.	**SAVE**
G	No room for line Im Speicher ist nicht genug Platz, um die neue Programmzeile aufzunehmen.	Bei Eingabe einer Zeile ins Programm.
H	STOP in INPUT **INPUT**-Daten haben mit **STOP** begonnen oder es wurde – für **INPUT LINE** – gedrückt. Im Gegensatz zu Meldung 9 verhält sich **CONTINUE** normal und wiederholt die **INPUT**-Anweisung.	**INPUT**
I	FOR without NEXT Eine **FOR**-Schleife sollte Null mal ausgeführt werden (etwa **FOR** n=1 **TO** 0), und die entsprechende **NEXT**-Anweisung konnte nicht gefunden werden.	**FOR**
J	Invalid I/O device	Arbeiten mit Microdrive, etc.
K	Invalid colour Die angegebene Zahl ist kein angemessener Wert.	**INK, PAPER, BORDER, FLASH, BRIGHT, INVERSE, OVER**; auch nach einem der Steuerzeichen dazu.
L	BREAK into program **BREAK** gedrückt, was zwischen zwei Anweisungen entdeckt wurde. Die Zeilen- und Anweisungsnummer in der Meldung beziehen sich auf die Anweisung vor **BREAK**, aber **CONTINUE** geht zur Anweisung danach (läßt aber Sprünge geschehen), so daß es keine Anweisung wiederholt.	Jede

Code	*Bedeutung*	*Situation*
M	RAMTOP no good Die für RAMTOP angegebene Zahl ist entweder zu groß oder zu klein.	**CLEAR**; möglicherweise bei **RUN**
N	Statement lost Sprung zu einer Anweisung, die nicht mehr existiert.	**RETURN, NEXT, CONTINUE**
O	Invalid stream	Arbeiten mit Microdrive, etc.
P	FN without DEF bei benutzergewählter Definition	**FN**
Q	Parameter error Falsche Zahl von Argumenten, oder eines ist von der falschen Art (String statt Zahl oder umgekehrt).	**FN**
R	Tape loading error Auf dem Band ist eine Datei gefunden worden, die aber aus irgendeinem Grund nicht eingelesen werden konnte oder sich nicht vergleichen ließ.	**VERIFY, LOAD** oder **MERGE**

Eine Beschreibung des ZX Spectrum zum Nachschlagen

Der erste Abschnitt dieses Anhangs wiederholt diesen Teil der Anleitung für Tastatur und Bildschirm.

Die Tastatur

ZX Spectrum-Zeichen umfassen nicht nur die einzelnen *Symbole* (Buchstaben, Ziffern, usw.) sondern auch die zusammengesetzten *Token* (Schlüsselwörter, Funktionsnamen, usw.). Letztere werden mit Tastendruck eingegeben und nicht Buchstabe für Buchstabe geschrieben. Um alle diese Funktionen und Befehle zu erhalten, haben manche Tasten bis zu fünf und mehr verschiedene Bedeutungen. Sie werden zum einen Teil dadurch gegeben, daß man umschaltet (das heißt, entweder die Taste **CAPS SHIFT** oder die Taste **SYMBOL SHIFT** gleichzeitig mit der betreffenden Taste drückt), und zum anderen dadurch, daß der Computer in verschiedenen Betriebsarten ist (*Modus*).

Der Modus wird angezeigt durch den *Cursor*, einen blinkenden Buchstaben. Er läßt erkennen, wo das nächste Zeichen von der Tastatur eingefügt werden wird.

Der K (für **keyword** = Schlüsselwort)-Modus tritt an die Stelle des L-Modus, wenn der Computer einen Befehl oder eine Programmzeile erwartet (also keine **INPUT**-Daten) und aus seiner Position auf der Zeile weiß, daß er eine Zeilennummer oder ein Schlüsselwort erhalten soll. Das ist am Zeilenanfang oder gleich nach **THEN** oder gleich nach **:** (außer in einem String). Ohne Umschaltung wird die nächste Taste entweder als ein Schlüsselwort (es steht auf der Taste) oder als eine Ziffer aufgefaßt.

In der Regel herrscht sonst immer der L (für letter = Buchstabe)-Modus. Ohne Umschaltung wird die nächste Taste als das Hauptsymbol auf dieser Taste, bei Buchstaben in Kleinschrift, aufgefaßt.

Sowohl im K- als auch im L-Modus werden **SYMBOL SHIFT** zusammen mit einer anderen Taste aufgefaßt als das rote Nebenzeichen auf der Taste, **CAPS SHIFT** zusammen mit einer Zifferntaste als die weiß geschriebene Steuerfunktion über der Taste. **CAPS SHIFT** zusammen mit anderen Tasten wirkt nicht auf die Schlüsselwörter im K-Modus, im L-Modus wandelt es kleine in große Buchstaben um.

Der C (für capitals = Großbuchstabe)-Modus ist eine Abart des L-Modus. Alle Buchstaben erscheinen hier groß geschrieben. **CAPS LOCK** bewirkt eine Umschaltung vom L- zum C-Modus und zurück.

Der E (für extended = erweitert)-Modus dient dazu, weitere Zeichen zu erhalten, zumeist Token. Er tritt ein, nachdem beide Umschalttasten gleichzeitig gedrückt worden sind, und hält nur einen Tastendruck lang an. In diesem Modus liefert ein Buchstabe ein Zeichen oder Token (in grüner Schrift über der Taste), wenn nicht umgeschaltet wird, und ein anderes (in roter Schrift unter der Taste), wenn dazu eine der Umschalttasten betätigt wird. Eine Zifferntaste liefert dann ein Token, wenn sie zusammen mit **SYMBOL SHIFT** gedrückt wird; im anderen Fall erscheint eine Farbsteuerungssequenz.

Der G (für graphics = Grafik)-Modus tritt ein, nachdem **GRAPHICS** (**CAPS SHIFT** und **9**) gedrückt wurde, und hält an, bis es wieder gedrückt wird. Eine Zifferntaste liefert ein Grafikmosaik (ausgenommen **GRAPHICS** oder **DELETE**), und jede Buchstaben-

taste mit Ausnahme von V, W, X, Y und Z eine benutzergewählte Grafik.

Jede Taste, die länger als 2 oder 3 Sekunden niedergedrückt wird, wiederholt automatisch.

Eingabe über die Tastatur erscheint, wie geschrieben, auf der unteren Bildschirmhälfte. Jedes Zeichen (Einzelsymbol oder zusammengesetztes Wort) wird unmittelbar vor dem Cursor eingeschoben. Der Cursor kann mit **CAPS SHIFT** und **5** nach links, mit **CAPS SHIFT** und **8** nach rechts verschoben werden. Das Zeichen vor dem Cursor läßt sich durch **DELETE** (**CAPS SHIFT** und **0**) löschen. (Merke: Die ganze Zeile kann gelöscht werden durch **EDIT** (**CAPS SHIFT** und **1**), gefolgt von **ENTER**.)

Bei Druck auf **ENTER** wird die Zeile, je nach dem, was angemessen ist, ausgeführt, ins Programm genommen oder als **INPUT** verwendet, es sei denn, sie enthält einen Syntaxfehler. In diesem Fall erscheint vor dem Fehler ein blinkendes **?**

Mit der Eingabe von Programmzeilen wird auf der oberen Bildschirmhälfte ein Listing angezeigt. Die Art, wie das Listing produziert wird, ist ziemlich kompliziert und wird in Kapitel 2 näher erläutert. Die zuletzt eingegebene Zeile heißt *laufende* Zeile und wird angezeigt durch das Symbol >. Das läßt sich aber ändern durch die Tasten ◆ (**CAPS SHIFT** und **6**) und ◆ (**CAPS SHIFT** und **7**). Bei Druck auf **EDIT** (**CAPS SHIFT** und **1**) wird die laufende Zeile zur unteren Bildschirmhälfte heruntergeholt und kann dort redigiert werden.

Wird ein Befehl ausgeführt oder ein Programm gefahren, erscheint die Ausgabe in der oberen Bildschirmhälfte und bleibt dort, bis eine Programmzeile eingegeben oder mit einer Leerzeile **ENTER** gedrückt oder die Tasten ◆ oder ◆ betätigt werden. In der unteren Hälfte erscheint eine Meldung. Sie nennt einen Code (Ziffer oder Buchstabe), der Sie auf Anhang B verweist, faßt kurz zusammen, was in Anhang B darüber steht, und gibt die Nummer der Zeile mit der zuletzt ausgeführten Anweisung (bei einem Befehl 0) und die Position der Anweisung innerhalb der Zeile an. Die Meldung bleibt auf dem Bildschirm, bis eine Taste gedrückt wird (und den K-Modus anzeigt).

Unter bestimmten Umständen wirkt **CAPS SHIFT** zusammen mit der **SPACE**-Taste als **BREAK** und hält den Computer mit Meldung **D** oder **L** an. Das wird erkannt

(I) am Ende einer Anweisung, während ein Programm läuft, oder
(II) während der Computer mit Kassettenrecorder oder Drucker läuft.

Der Fernseh-Bildschirm

Er hat 24 Zeilen zu je 32 Zeichen und besteht aus zwei Teilen. Der obere Teil umfaßt im Höchstfall 22 Zeilen und zeigt entweder ein Listing oder eine Programmausgabe. Wenn die Anzeige im oberen Teil am unteren ankommt, rollt sie um eine Zeile nach oben; würde das bedeuten, daß eine Zeile verlorengeht, die Sie noch nicht sehen konnten, hält der Computer mit der Meldung **scroll?** (abrollen?) an. Ein Druck auf die Tasten **N, SPACE** oder **STOP** hält das Programm an mit der Meldung **D BREAK – CONT repeats** (Unterbrechung – Wiederholung mit **CONTINUE**); jede andere Taste setzt das Abrollen fort. Der untere Bildschirmteil wird verwendet für die Eingabe von Befehlen, Programmzeilen und **INPUT**-Daten, sowie für die Anzeige von Meldungen. Der untere Teil beginnt mit zwei Zeilen (die obere bleibt leer), dehnt sich aber aus, um aufzunehmen, was eingetippt wird. Wird die laufende Anzeigeposition in der oberen Hälfte erreicht, führt weitere Ausdehnung dazu, daß die obere Hälfte abrollt.

Jede Zeichenposition hat *Attribute*, die ihre *Paper* (=Hintergrund)- und *Ink* (=Vordergrund)-Farbe bezeichnen, eine zweistufige Helligkeit und ob sie blinkt oder nicht. Die verfügbaren Farben sind black, blue, red, magenta, green, cyan, yellow und white (schwarz, blau, rot, violett, grün, hellblau, gelb und weiß.)

Der Rand des Bildschirms kann mit der **BORDER**-Anweisung auf jede dieser Farben gesetzt werden.

Eine Zeichenposition ist aufgeteilt in 8 mal 8 Pixel. Hochauflösende Grafik wird erzielt, indem man die Pixel einzeln so setzt, daß sie entweder die Ink- oder die Paperfarbe für diese Zeichenposition zeigen.

Die Attribute an einer Zeichenposition werden angepaßt, sobald dort ein Zeichen geschrieben oder ein Pixel mit **PLOT** angesprochen wird. Die genaue Art der Anpassung wird bestimmt durch die *Anzeigeparameter*, von denen es zwei Sätze (*permanent* und *zeitweilig*) zu je sechs gibt: die Parameter **PAPER, INK, FLASH, BRIGHT, INVERSE** und **OVER.** Dauerparameter für den oberen Bildschirmteil werden gesetzt durch Anweisungen **PAPER, INK** etc., und bleiben bis auf weiteres bestehen. (Anfänglich sind sie Ink schwarz auf Paper weiß, bei normaler Helligkeit, ohne Blinken, Video normal und ohne Überschreiben). Dauerparameter für den unteren Bildschirmteil verwenden die Borderfarbe als Paperfarbe, mit einer schwarzen oder weißen Inkfarbe als Kontrast, normale Helligkeit, kein Blinken, Video normal und kein Überschreiben.

Zeitweilige Parameter werden gesetzt von Posten **PAPER, INK** usw., die in **PRINT**-, **LPRINT**-, **INPUT**-, **PLOT**-, **DRAW**-, und **CIRCLE**-Anweisungen stehen, und ferner von Steuerzeichen **PAPER, INK** etc., wenn sie dem Fernseher zugeführt werden – ihnen folgt ein zusätzliches Byte, das den Parameterwert angibt. Zeitweilige Parameter gelten nur bis zum Ende der **PRINT** (oder sonstigen)-Anweisung, oder, bei **INPUT**-Anweisungen, bis von der Tastatur **INPUT**-Daten verlangt werden, worauf an ihre Stelle Dauerparameter treten.

PAPER- und **INK**-Parameter liegen zwischen 0 und 9. Die Parameter 0 bis 7 sind die verwendeten Farben beim Anzeigen eines Zeichens:

0 black	= schwarz
1 blue	= blau
2 red	= rot
3 magenta	= violett
4 green	= grün
5 cyan	= hellblau
6 yellow	= gelb
7 white	= weiß

Parameter 8 ('durchsichtig') bestimmt, daß die Farbe auf dem Bildschirm unverändert bleiben soll, wenn ein Zeichen angezeigt wird.

Parameter 9 ('Kontrast') bestimmt, daß die betreffende Farbe (Paper oder Ink) entweder schwarz oder weiß werden soll, um sich von der anderen Farbe abzuheben.

Die Parameter **FLASH** und **BRIGHT** sind 0, 1 oder 8: 1 bedeutet, daß Blinken oder starke Helligkeit eingeschaltet, 0, daß sie abgeschaltet ist, und 8 ('durchsichtig'), daß sie bei jeder Zeichenposition unverändert bleiben.

Die Parameter für **OVER** und **INVERSE** sind 0 oder 1.

OVER Ø neue Zeichen löschen alte
OVER 1 Die Bitmuster der alten und neuen Zeichen werden mit einer 'exclusive or'-Operation (*Überschreiben*) kombiniert.
INVERSE Ø neue Zeichen werden angezeigt in Inkfarbe auf Paperfarbe (Video normal)
INVERSE 1 neue Zeichen werden angezeigt in Paperfarbe auf Inkfarbe (Negativschrift)

Erhält der Fernseher ein **TAB**-Steuerzeichen, werden zwei zusätzliche Bytes erwartet, die einen Tabstop n bestimmen (das weniger bedeutsame Byte zuerst). Dabei wird modulo 32 auf (z.B.) n_0 reduziert, dann werden so viele Leerstellen angezeigt, daß die Anzeigeposition in die Spalte n_0 gelangt.

Kommt ein Komma-Steuerzeichen, werden so viele Leerstellen (mindestens eine) angezeigt, daß die Anzeigeposition in Spalte Ø oder Spalte 16 gelangt.

Erhält der Fernseher ein **ENTER**-Steuerzeichen, wird die Anzeigeposition auf die nächste Zeile versetzt.

Der Drucker

Ausgabe zum Drucker erfolgt über einen Puffer von einer Zeile Länge (32 Zeichen). Eine Zeile wird zum Drucker geschickt,

(I) wenn die Anzeige von einer Zeile zur nächsten überläuft,
(II) wenn ein **ENTER**-Zeichen empfangen wird,
(III) am Ende eines Programms, wenn noch etwas ungedruckt geblieben ist,
(IV) wenn ein Steuerzeichen **TAB** oder Komma die Druckposition zu einer neuen Zeile rückt.

TAB- und Komma-Steuerzeichen geben Leerräume in derselben Weise aus wie auf dem Bildschirm.

Die **AT**-Steuerung verändert die Druckposition mit der Spaltennummer und beachtet die Zeilennummer nicht.

Der Drucker wird von **INVERSE**- und **OVER**-Steuerungen (und auch Anweisungen) ebenso betroffen wie der Bildschirm, aber nicht von **PAPER**, **INK**, **FLASH** oder **BRIGHT**.

Der Drucker bleibt mit Fehlermeldung B stehen, wenn **BREAK** gedrückt wird.

Ist der Drucker nicht angeschlossen, geht die Ausgabe einfach verloren.

BASIC

Zahlen werden gespeichert bis zu einer Genauigkeit von 9 oder 10 Stellen. Die größte Zahl, die Sie erreichen können, liegt bei etwa 10^{38}, die kleinste (positive) Zahl bei $4*10^{-39}$.

Eine Zahl wird im ZX Spectrum in Fließ- oder Gleitpunkt-Binärschreibweise gespeichert, mit einem Exponentenbyte e ($1<=e<=255$) und vier Mantissenbytes m ($\frac{1}{2}<=m<1$). Das stellt die Zahl $m*2^{e-128}$ dar.

Da $\frac{1}{2}<m=<1$, ist das bedeutsamste Bit der Mantisse m stets 1. Demnach können wir es faktisch ersetzen durch ein Bit mit dem Zeichen 0 für positive Zahlen und 1 für negative.

Kleine ganze Zahlen werden auf besondere Weise dargestellt. Dabei ist das erste Byte 0, das zweite ein Vorzeichenbyte (0 oder FFh), und das dritte und vierte sind die ganze Zahl in Zweierkomplementärform, voran das weniger bedeutsame Byte.

Numerische Variable haben Namen von willkürlicher Länge, beginnend mit einem Buchstaben, fortgesetzt mit Buchstaben und Ziffern. Leerstellen und Farbsteuerungen werden ignoriert, alle Buchstaben in Kleinbuchstaben umgewandelt.

Steuervariable für **FOR-NEXT**-Schleifen haben Namen, die einen Buchstaben lang sind.

Numerische Arrays haben Namen von der Länge eines Einzelbuchstaben, der dem Namen einer einfachen Variablen entsprechen kann. Sie können beliebig viele Dimensionen von willkürlicher Größe haben. Indices beginnen bei 1.

Strings sind in der Länge beliebig. Der Name eines Strings besteht aus einem einzelnen Buchstaben, gefolgt von $.

Stringarrays können beliebig viele Dimensionen von willkürlicher Größe haben. Der Name ist ein Einzelbuchstabe, gefolgt von $, und darf nicht derselbe sein wie der Name eines Strings. Alle Strings in einem Array haben dieselbe festgelegte Länge, die in der **DIM**-Anweisung als eine zusätzliche, abschließende Dimension festgelegt wird. Indices beginnen bei 1.

Slicing: Substrings von Strings können mit Hilfe von *Slicers* bestimmt werden. Ein Slicer kann

(I) leer sein
oder
(II) ein numerischer Ausdruck
oder
(III) beliebiger numerischer Ausdruck **TO** beliebiger numerischer Ausdruck

und wird dazu verwendet, einen Substring auszudrücken entweder durch
(a) einen Stringausdruck (Slicer)
oder
(b) durch eine Stringarray-Variable (Index, ..; Index, Slicer).

In (a) sei unterstellt, daß der Stringausdruck den Wert s$ hat.

Wenn der Slicer leer ist, dann ist das Ergebnis s$, betrachtet als ein Substring seiner selbst.

Wenn der Slicer ein Substring mit dem Wert m ist, dann ist das Ergebnis das m-te

Zeichen von s$ (ein Substring von der Länge 1).

Wenn der Slicer die Form (III) hat, sei unterstellt, daß der erste numerische Ausdruck den Wert m hat (der Vorgabewert ist 1), und der zweite n (der Vorgabewert ist die Länge von s$).

Wenn 1<=m<=n<= der Länge von s$, dann ist das Resultat der Substring von s$, beginnend mit dem m-ten Zeichen und aufhörend mit dem n-ten.

Ist Ø<=n<m, dann ist die Folge der leere String.

Ansonsten kommt es zur Fehlermeldung 3.

Das Slicing wird ausgeführt, bevor Funktionen oder Operationen behandelt werden, es sei denn, Klammern verlangen etwas anderes.

Substrings kann etwas zugeordnet werden (siehe **LET**).

Wenn in ein Stringliteral ein Stringanführungszeichen gesetzt werden soll, muß es verdoppelt werden.

Funktionen

Das Argument einer Funktion benötigt keine Klammern, wenn es eine Konstante oder eine (möglicherweise indizierte oder durch Slicing gewonnene) Variable ist.

Funktion	*Art des Arguments (x)*	*Resultat*
ABS	Zahl	Absolute Größe
ACS	Zahl	Arkkosinus in Radianten. Fehler A, wenn x nicht im Bereich –1 bis +1.
AND	binäre Operation, rechter Operand stets eine Zahl. Numerischer linker Operand:	A **AND** B = { A wenn B<>Ø ; Ø wenn B=Ø
	Linker Operand als String:	A$ **AND** B = { A$ wenn B<>Ø ; "" wenn B=Ø
ASN	Zahl	Arkussinus in Radianten. Fehler A, wenn x nicht im Bereich –1 bis +1
ATN	Zahl	Arkustangens in Radianten
ATTR	zwei Argumente, x und y, beide Zahlen; in Klammern	Eine Zahl, deren binäre Form die Attribute von Zeile x, Spalte 7 auf dem Bildschirm codiert. Bit 7 (am bedeutsamsten) ist 1 für Blinken, Ø für nicht Blinken. Bit 6 ist für hell, Ø für normal. Die Bits 5 bis 3 sind die Paperfarbe, Bits 2 bis Ø die Inkfarbe. Fehler Ø, wenn nicht Ø<=x<=23 und Ø<=y<=31
BIN		Das ist eigentlich keine Funktion, sondern eine andere Schreibweise für Zahlen: BIN gefolgt von einer Folge von Nullen und Einsen ist die Zahl mit einer solchen Darstellung in binärer Schreibweise.

Funktion	*Art des Arguments*	*Resultat*
CHR$	Zahl	Das Zeichen, dessen Code x ist, abgerundet zur nächsten ganzen Zahl
CODE	String	Der Code des ersten Zeichens in x (oder Ø, wenn x der leere String ist)
COS	Zahl (in Radianten)	Kosinus x
EXP	Zahl	e^x
FN		FN, gefolgt von einem Buchstaben, ruft eine benutzergewählte Funktion auf (siehe **DEF**). Die Argumente müssen in Klammern stehen; auch ohne Argumente müssen die Klammern vorhanden sein.
IN	Zahl	Das Resultat der Eingabe auf Prozessorebene von Baustein x (Ø<=x<=FFFFh). (Lädt das bc-Registerpaar mit x und führt den Assemblerbefehl in a(c) aus)
INKEY$	keines	Liest die Tastatur. Die Folge ist das Zeichen (im **L**- oder **C**-Modus), das der gedrückten Taste entspricht, wenn es genau eines ist, sonst der leere String.
INT	Zahl	Integerteil (ganze Zahl). Rundet stets ab.
LEN	String	Länge
LN	Zahl	Natürlicher Logarithmus (auf Basis e). Fehler A, wenn x<=Ø
NOT	Zahl	Ø, wenn x<>Ø, 1, wenn x=Ø. **NOT** hat Priorität 4
OR	binäre Operation, beide Operanden Zahlen	a **OR** b = { 1, wenn b<>Ø ; a, wenn b=Ø } **OR** hat Priorität 2
PEEK	Zahl	Der Wert des Speicherbyte, dessen Adresse x ist (abgerundet zur nächsten ganzen Zahl). Fehler B, wenn x nicht im Bereich Ø bis 65535
PI	keines	π (3.14159265 ...)
POINT	Zwei Argumente, x und y, beide Zahlen; in Klammern	1, wenn das Pixel bei (x,y) Inkfarbe hat, Ø, wenn Paperfarbe Fehler B, wenn nicht Ø<=x<=255 und Ø<=y<=175
RND	keines	Die nächste pseudozufällige Zahl in einer Sequenz, generiert durch die Potenzen von 75 modulo 65537, 1 abgezogen und durch 65536 geteilt. Ø<y<1

Funktion	*Art des Arguments*	*Resultat*
SCREEN$	Zwei Argumente, x und y, beide Zahlen; in Klammern	Das Zeichen, das normal oder in Negativschrift in Zeile x, Spalte y auf dem Bildschirm erscheint. Ergibt einen leeren String, wenn das Zeichen nicht erkannt wird. Fehler B, wenn nicht 0<=x<=23 und 0<=y<=31
SGN	Zahl	Signum: Das Vorzeichen (–1 für negativ, 0 für Null oder +1 für positiv) von x
SIN	Zahl (in Radianten)	Sinus x
SQR	Zahl	Quadratwurzel. Fehler A, wenn x<0
STR$	Zahl	Der Zeichenstring, der zur Anzeige käme, wenn x angezeigt würde
TAN	Zahl (in Radianten)	Tangens
USR	Zahl	Ruft die Maschinencode-Subroutine auf, deren Startadresse x ist. Beim Rücksprung ist das Resultat der Inhalt des Registerpaars bc.
USR	String	Die Adresse des Bitmusters für die benutzergewählte Grafik, die x entspricht. Fehler A, wenn x kein einzelner Buchstabe zwischen a und u oder eine benutzergewählte Grafik
VAL	String	Behandelt x (ohne seine begrenzenden Anführungszeichen) als einen numerischen Ausdruck. Fehler C, wenn x einen Syntaxfehler enthält oder einen numerischen Wert angibt. Andere Fehler möglich, je nach Ausdruck
VAL$	String	Behandelt x (ohne seine begrenzenden Anführungszeichen) als einen Stringausdruck. Fehler C, wenn x einen Syntaxfehler enthält oder einen numerischen Wert angibt. Andere Fehler möglich, wie bei **VAL**
–	Zahl	Negation

Die folgenden Zeichen sind binäre Operationen:

+	Addition (bei Zahlen) oder Verkettung (bei Strings)
–	Subtraktion
*	Multiplikation
/	Division
↑	Erhebung zur Potenz. Fehler B, wenn linker Operand negativ

=	ist gleich	beide Operanden müssen von gleicher Art sein. Das Resultat ist eine Zahl 1, wenn der Vergleich gilt, und Ø, wenn das nicht zutrifft
>	größer als	
<	kleiner als	
<=	kleiner als oder gleich	
>=	größer als oder gleich	
<>	nicht gleich	

Funktionen und Operationen haben folgende Prioritäten:

Operation	*Priorität*
Indizieren und Slicing	12
Alle Funktionen außer **NOT** und unärem Minus	11
↑	10
unäres Minus (d.i. Minus, nur verwendet, um etwas zu negieren)	9
*, /	8
+, – (Minus, verwendet, um eine Zahl von einer anderen zu subtrahieren)	6
=, >, <, <=, >=, <>	5
NOT	4
AND	3
OR	2

Anweisungen

In der folgenden Liste stellen dar

α	einen einzelnen Buchstaben
v	eine Variable
x, y, z	numerische Ausdrücke
m, n	numerische Ausdrücke, die zur nächsten ganzen Zahl abgerundet werden
e	einen Ausdruck
f	einen als String behandelten Ausdruck
s	eine Folge von Anweisungen, getrennt durch Doppelpunkte
c	eine Folge von Farbenposten, jeder abgeschlossen durch Kommas , oder Strichpunkte ;. ein Farbenposten hat die Form einer Anweisung **PAPER, INK, FLASH, BRIGHT, INVERSE** oder **OVER**.

Beachten Sie, daß beliebige Ausdrücke überall zugelassen sind (außer bei der Zeilennummer zu Beginn einer Anweisung).

Alle Anweisungen außer **INPUT, DEF** und **DATA** können verwendet werden entweder als Befehle oder in Programmen (obwohl sie in der einen Form vernünftiger sein können als in der anderen). Ein Befehl oder eine Programmzeile kann mehrere Anweisungen, getrennt durch Doppelpunkte (:), enthalten. Es gibt keine Einschränkung dabei, wo in einer Zeile eine bestimmte Anweisung stehen darf – siehe auch **IF** und **REM**.

BEEP x, y
Läßt aus dem Lautsprecher für x Sekunden bei einer Tonhöhe y über dem mittleren C (oder darunter, wenn y negativ) einen Ton hören.

BORDER m
Setzt die Farbe der Bildschirmumrandung und die Paperfarbe für den unteren Bildschirmteil.
Fehler K, wenn m nicht im Bereich Ø bis 7.

BRIGHT
Setzt Helligkeit von Zeichen, die anschließend angezeigt werden. n=Ø für normal, 1 für hell, 8 für durchsichtig.
Fehler K, wenn n nicht Ø, 1 oder 8

CAT
Wirkt nicht ohne Microdrive etc.

CIRCLE x, y, z
Zeichnet Bogen eines Kreises, Mittelpunkt (x,y), Radius z

CLEAR
Löscht alle Variablen, macht den Platz frei, den sie besetzt hatten.
Bewirkt **RESTORE** und **CLS**, setzt die **PLOT**-Position an die linke untere Ecke zurück und löscht den **GO SUB**-Stapel.

CLEAR n
Wie **CLEAR**, verändert aber, wenn möglich, die Systemvariable **RAMTOP** zu n und setzt dort den neuen **GO SUB**-Stapel

CLOSE #
Wirkt nicht ohne Microdrive, etc.

CLS
(Clear Screen). Löscht die Displaydatei.

CONTINUE
Setzt das Programm fort und beginnt dort, wo es beim letztenmal mit einer anderen Meldung als Ø aufgehört hat. Wenn die Meldung 9 oder Ø war, fährt es fort mit der nächsten Anweisung (wobei es Sprünge berücksichtigt); sonst wiederholt es diejenige, wo der Fehler eingetreten ist.
Stand die letzte Meldung in einer Befehlszeile, versucht **CONTINUE** die Befehlszeile fortzusetzen und geht entweder in eine Schleife, wenn der Fehler in Ø:1 war, gibt Meldung Ø, wenn er in Ø:2 war, oder Fehler N, wenn es Ø:3 oder höher war.
CONTINUE erscheint als **CONT** auf der Tastatur

COPY
Schickt eine Kopie der oberen 22 Displayzeilen zum Drucker, wenn dieser angeschlossen ist; im anderen Fall geschieht nichts. Beachten Sie: **COPY** kann nicht dazu verwendet werden, die automatischen Listings zu drucken, die auf dem Bildschirm erscheinen.
Meldung D, wenn **BREAK** gedrückt wird

DATA $e_1, e_2, e_3, \ldots$
Teil der **DATA**-Liste. Muß in einem Programm stehen

DEF FN $\alpha(n_1,\ldots,\alpha_k)$=e
Benutzergewählte Funktionsdefinition; muß in einem

Programm stehen. a und a_1 bis a_k ist entweder ein einzelner Buchstabe oder ein einzelner Buchstabe, gefolgt von '$' für Stringargument oder Resultat.

Hat die Form **DEF FN** a()=e, wenn keine Argumente

DELETE f — Wirkt nicht ohne Microdrive, etc.

DIM $\alpha(n_1,...,n_k)$ — Löscht jedes Array mit dem Namen a und setzt ein Array a von Zahlen mit k Dimensionen $n_1,...,n_k$. Initialisiert alle Werte auf Ø.

DIM $\alpha\$(n_1,...;n_k)$ — Löscht alle Arrays oder Strings mit dem Namen a$ und setzt ein Array von Zeichen mit k Dimensionen $n_1,..;n_k$. Initialisiert alle Werte auf " ". Das kann betrachtet werden als ein Array von Strings mit festgelegter Länge n_k, mit k–1 Dimensionen $n_1,..;n_{k-1}$.

Fehler 4 tritt auf, wenn kein Platz ist, um das Array unterzubringen. Ein Array ist undefiniert, bis es in einer **DIM**-Anweisung dimensioniert wird.

DRAW x,y — **DRAW** x,y,Ø

DRAW x,y,z — Zeichnet eine Linie von der laufenden Plotposition und bewegt x horizontal und y vertikal dazu, während es einen Winkel z durchläuft.

Fehler B, wenn sie vom Bildschirm läuft.

ERASE — Wirkt nicht ohne Microdrive, etc.

FLASH — Bestimmt, welche Zeichen blinken oder gleichbleiben. n=Ø für gleichbleibend, n=1 für Blinken, n=8 ohne Veränderung.

FOR α=x **TO** y — **FOR** a=x **TO** y **STEP** 1

FOR α=x **TO** y **STEP** z — Löscht jede einfache Variable a und setzt eine Steuervariable mit Wert x, Limit y, Step z, und Schleifenadresse, die sich auf die Anweisung nach der **FOR**-Anweisung bezieht. Prüft, ob der Anfangswert größer ist (wenn Step>=Ø) oder kleiner (wenn Step<Ø) als das Limit, und springt in diesem Fall zu Anweisung **NEXT** a, liefert aber Fehler 1, wenn es sie nicht gibt. Siehe **NEXT**.

Fehler 4 tritt auf, wenn für die Steuervariable kein Platz ist.

FORMAT f — Wirkt nicht ohne Microdrive, etc.

GO SUB n — Schiebt die Zeilennummer der **GO SUB**-Anweisung auf einen Stapel; dann wie **GO TO** n.

Fehler 4 kann auftreten, wenn es nicht genug **RETURN**-Anweisungen gibt

GO TO n — Springt zu Zeile n (oder, wenn diese nicht vorhanden, zur ersten Zeile danach)

IF x **THEN** s

Wenn x wahr (nicht Null), dann wird s ausgeführt. Beachten Sie, daß s alle Anweisungen bis zum Ende der Zeile umfaßt. Die Form '**IF** x **THEN** Zeilennummer' ist nicht zugelassen

INK n

Setzt die Ink(Vordergrund)-Farbe von Zeichen, die anschließend angezeigt werden. n ist im Bereich Ø bis 7 für eine Farbe, n = 8 für durchsichtig oder 9 für Kontrast. Siehe *Der Fernseh-Bildschirm*, Anhang B.

Fehler K, wenn n nicht im Bereich Ø bis 9.

INPUT ...

Das '...' ist eine Folge von **INPUT**-Posten, wie in einer **PRINT**-Anweisung, getrennt durch Kommas, Strichpunkte oder Apostrophe. Ein **INPUT**-Posten kann sein

(I) jeder **PRINT**-Posten, der nicht mit einem Buchstaben beginnt
(II) ein Variablenname, oder
(III) **LINE**, dann ein Variablenname vom Stringtyp.

Die **INPUT**-Posten und Trennsymbole in (I) werden genauso behandelt wie bei **PRINT**, nur wird alles im unteren Bildschirmteil angezeigt.

Bei (II) hält der Computer an und wartet auf die Eingabe eines Ausdrucks von der Tastatur; dieser Wert wird der Variablen zugeteilt. Die Eingabe wird auf die übliche Weise angezeigt, Syntaxfehler führen zum blinkenden ?. Bei Ausdrücken vom Stringtyp wird der Eingabepuffer darauf initialisiert, zwei Stringanführungszeichen zu enthalten (die notfalls gelöscht werden können). Ist das erste Zeichen der Eingabe **STOP**, bleibt das Programm mit Fehler H stehen. (III) ist wie (II), außer, daß der Input als ein Stringliteral ohne Anführungszeichen behandelt wird und der **STOP**-Mechanismus nicht funktioniert; zum Anhalten müssen Sie statt dessen ⬅ drücken.

INVERSE n

Steuert Inversion (Umstellung auf Negativschrift) von Zeichen, die anschließend angezeigt werden. Wenn n = Ø, werden Zeichen in *True Video* angezeigt, als Inkfarbe auf Paperfarbe.

Wenn n = 1, werden Zeichen in *Inverse Video*, d.h. Paperfarbe auf Inkfarbe angezeigt. Siehe *Der Fernseh-Bildschirm*, Anhang B.

Fehler K, wenn n nicht Ø oder 1 ist

LET v=e

Teilt der Variablen v den Wert e zu. **LET** kann nicht weggelassen werden. Eine einfache Variable ist undefiniert, bis sie in einer **LET, READ**- oder **INPUT**-Anweisung zugeteilt wird. Wenn v eine indizierte Stringvariable oder eine durch Slicing entstandene Stringvariable ist (Substring), dann erfolgt die Zuteilung nach dem *Prokru-*

stes-Prinzip (festgelegte Länge): Der Stringwert von e wird entweder gekürzt oder rechts mit Leerstellen aufgefüllt, damit er dieselbe Länge hat wie die Variable v.

LIST — **LIST** 0

LIST n — Listet das Programm auf dem oberen Bildschirmteil auf, beginnt mit der ersten Zeile, deren Nummer mindestens n ist, und macht n zur laufenden Zeile

LLIST — **LLIST** 0

LLIST n — Wie **LIST**, benützt aber den Drucker

LOAD f — Lädt Programme und Variable

LOAD f **DATA** () — Lädt ein numerisches Array

LOAD f **DATA** $() — Lädt Zeichenarray $

LOAD f **CODE** m,n — Lädt höchstens n Bytes, beginnt bei Adresse m

LOAD f **CODE** m — Lädt Bytes, beginnt bei Adresse m

LOAD f **CODE** — Lädt Bytes zurück zu der Adresse, von der sie gesichert wurden

LOAD f **SCREEN$** — **LOAD** f **CODE** 16384,6912.
Sucht nach Datei der richtigen Art auf Bandkassette und lädt sie, wobei vorherige Versionen im Speicher gelöscht werden. Siehe Kapitel 20

LPRINT — Wie **PRINT**, benützt aber den Drucker

MERGE f — Wie **LOAD** f, löscht aber nicht alte Programmzeilen, es sei denn, um Platz für neue mit derselben Zeilennummer oder demselben Namen zu schaffen

MOVE f_1,f_2 — Wirkt nicht ohne Microdrive, etc.

NEW — Startet das BASIC-System von neuem, löscht Programm und Variable, benützt Speicher und Variable bis zu dem Byte, dessen Adresse in der Systemvariablen RAMTOP ist (dieses eingeschlossen), und beläßt die Systemvariablen UDG, P RAMT, RASP und PIP

NEXT a — (I) Findet die Steuervariable
(II) Addiert seine Schrittfolge, ihren Wert
(III) Wenn Step>=0 und Wert>das Limit, oder wenn Step<0 und Wert<das Limit, springt es zur Schleifenanweisung.
Fehler 2, wenn keine Variable a vorhanden
Fehler 1, wenn zwar vorhanden, aber nicht Steuervariable

OPEN # — Wirkt nicht ohne Mikrodrive, etc.

OUT m,n — Gibt auf Prozessorebene am Port m n Bytes aus. (Lädt das Registerpaar bc mit m, das Register a mit n, und führt den Assemblerbefehl aus: out (c),a.)

0<=m<=65535, -255<=n<=255, sonst Fehler B

OVER n — Steuert Überschreiben für Zeichen, die anschließend angezeigt werden.

Wenn n=0, löschen Zeichen vorherige Zeichen an dieser Position.

Wenn n=1, werden neue Zeichen mit alten Zeichen gemischt, um Inkfarbe zu zeigen, wo eines von beiden (aber nicht beide) Inkfarbe hatte, und Paperfarbe, wenn sie beide Paper- oder beide Inkfarbe hatten. Siehe *Der Fernseh-Bildschirm,* Anhang B.

Fehler K, wenn n nicht 0 oder 1

PAPER n — Wie **INK**, steuert aber die Paper(Hintergrund)-Farbe

Pause n — Unterbricht den Computer und zeigt für n Einzelbilder die Displaydatei (bei 50 Bildern pro Sekunde, in Amerika bei 60) oder bis eine Taste gedrückt wird.
0<=n<=65535, sonst Fehler B. Wenn n=0, wird die Pause nicht gezählt, sondern dauert an, bis eine Taste gedrückt wird

PLOT c;m,n — Zeigt einen Inkpunkt (abhängig von **OVER** und **INVERSE**) am Pixel (|m|,|n|) an; versetzt die **PLOT**-Position.

Falls die Farbenposten c nichts anderes festlegen, wird die Inkfarbe an der Zeichenposition, die das Pixel enthält, zur laufenden permanenten Inkfarbe verändert, die anderen (Papierfarbe, Blinken und Helligkeit) bleiben unberührt.
0<= |m| <=255, 0<= |n| <=175, sonst Fehler B

POKE m,n — Schreibt den Wert n dem Speicherbyte mit Adresse m ein. 0<=m<=65535, -255<=n<=255, sonst Fehler B

PRINT ... — Das '...' ist eine Folge von **PRINT**-Posten, getrennt durch Kommas **,** Strichpunkte **;** oder Apostrophe **'**, und wird für die Ausgabe zum Fernseher in die Displaydatei geschrieben.

Ein Strichpunkt **;** zwischen zwei Posten hat keine Wirkung: Er dient nur dazu, die Posten zu trennen. Ein Komma **,** gibt das Komma-Steuerzeichen aus, ein Apostroph **'** das **ENTER**-Zeichen.

Am Ende der **PRINT**-Anweisung wird, wenn sie nicht mit einem Strichpunkt, einem Komma oder einem Apostroph aufhört, ein **ENTER**-Zeichen ausgegeben.

Ein **PRINT**-Posten kann sein

(I) leer, also nichts

(II) ein numerischer Ausdruck

Zuerst wird, wenn der Wert negativ ist, ein Minus-

zeichen angezeigt. Nun soll x der Wertmodul sein.

Wenn $x<=10^{-5}$ oder $x>=10^{13}$, dann wird es in wissenschaftlicher Schreibweise angezeigt. Der Mantissenteil hat bis zu acht Stellen (ohne nachlaufende Nullen), der Dezimalpunkt (er fehlt bei nur einer Ziffer) steht nach der ersten. Der Exponententeil ist E, gefolgt von + oder –, gefolgt von einer oder zwei Ziffern.

Andernfalls wird x in gewöhnlicher Dezimalschreibweise bis zu acht geltenden Ziffern angezeigt, nach dem Dezimalpunkt gibt es keine nachlaufenden Nullen. Einem Dezimalpunkt gleich zu Beginn folgt stets eine Null, also werden, zum Beispiel .Ø3 und Ø.3 in dieser Form angezeigt.

Ø wird angezeigt als Einzelziffer Ø.

(III) ein Stringausdruck

Die Token im String werden ausgedehnt, möglicherweise mit einer Leerstelle davor oder danach.

Steuerzeichen haben ihre Steuerwirkung.

Nicht erkannte Zeichen werden angezeigt als **?**.

(IV) **AT** m,n

Gibt ein **AT**-Steuerzeichen aus, gefolgt von einem Byte für m (die Zeilennummer) und einem Byte für n (die Spaltennummer).

(V) **TAB** n

Gibt ein **TAB**-Steuerzeichen aus, gefolgt von zwei Bytes für n (das weniger bedeutsame Byte zuerst), den **TAB**-Stop.

(VI) Ein Farbenposten, der die Form einer Weisung **PAPER, INK, FLASH, BRIGHT, INVERSE** oder **OVER** hat

RANDOMIZE

RANDOMIZE Ø

RANDOMIZE n

Setzt die Systemvariable (namens SEED), mit welcher der nächste Wert von **RND** generiert wird. Wenn n<>Ø, erhält SEED den Wert n; wenn n=Ø, dann den Wert einer anderen Systemvariablen (namens FRAMES), der die auf dem Bild bisher gezeigten Einzelbilder zählt und damit einigermaßen zufällig sein sollte.

RANDOMIZE erscheint auf der Tastatur als **RAND**.

Fehler B tritt auf, wenn n nicht im Bereich Ø bis 65535 liegt.

READ $v_1, v_2, \ldots v_k$

Teilt den Variablen zu und verwendet aufeinanderfolgende Ausdrücke in der **DATA**-Liste.

Fehler C, wenn ein Ausdruck von der falschen Art ist.

Fehler E, wenn die **DATA**-Liste erschöpft ist und noch Variablen zu lesen sind

REM ... Keine Wirkung. '...' kann jede Zeichenfolge außer **ENTER** sein. Das kann einschließen **:**, so daß nach der **REM**-Anweisung auf derselben Zeile keine Anweisungen möglich sind

RESTORE **RESTORE** 0

RESTORE n Setzt den **DATA**-Zeiger zurück auf die erste Anweisung in einer Zeile mit der Nummer mindestens n: Die nächste **READ**-Anweisung beginnt dort zu lesen

RETURN Nimmt eine Bezugnahme auf eine Anweisung vom **GO SUB**-Stapel und springt zu der Zeile danach.

Fehler 7 tritt auf, wenn auf dem Stapel keine Bezugnahme auf eine Anweisung. Sie haben im Programm einen Fehler; die **RETURN**-Anweisungen entsprechen nicht den **GO SUB**s.

RUN **RUN** 0

RUN n **CLEAR**, und dann **GO TO** n

SAVE f Sichert das Programm und die Variablen

SAVE f **LINE** Sichert das Programm und die Variablen so, daß nach dem Laden ein automatischer Sprung zum Beginn des Programms erfolgt

SAVE f **LINE** m Sichert das Programm und die Variablen so, daß nach dem Laden automatisch zu Zeile m gesprungen wird

SAVE f **DATA** () Sichert das numerische Array

SAVE f **DATA** $() Sichert das Zeichenarray $

SAVE f **CODE** m,n Sichert n Bytes, beginnend bei Adresse m

SAVE f **SCREEN$** **SAVE** f **CODE** 16384,6912.

Sichert Information auf Kassette und gibt ihr den Namen f.

Fehler F, wenn f leer oder Länge elf oder mehr. Siehe Kapitel 20

STOP Hält das Programm mit Meldung 9 an. **CONTINUE** setzt mit der darauf folgenden Anweisung fort

VERIFY Wie **LOAD**, außer, daß die Daten nicht nach RAM geladen, sondern mit dem verglichen werden, was dort schon vorhanden ist.

Fehler R, wenn einer der Vergleiche verschiedene Bytes zeigt

Programmbeispiele

Dieser Anhang enthält einige Programmbeispiele, mit denen die Fähigkeiten des ZX Spectrum vorgeführt werden sollen.
Hier das erste Programm: Man gibt ein Datum ein und bekommt den Wochentag genannt, auf den das Datum fiel (oder fällt):

```
 10 REM Wandelt Datum in Wochentag um
 20 DIM d$(7,10): REM Wochentage
 30 FOR n=1 TO 7: READ d$(n): NEXT n
 40 DIM m(12): REM Anzahl der Tage im Monat
 50 FOR n=1 TO 12: READ m(n): NEXT n
100 REM Datumseingabe
110 INPUT "Tag?"; Tag
120 INPUT "Monat"; Monat
130 INPUT "Jahr (nur 20. Jahrhundert)?"; Jahr
140 IF Jahr<1901 THEN PRINT "20. Jahrhundert beginnt 1901";
    GO TO 100
150 IF Jahr<2000 THEN PRINT "20. Jahrhundert endet bei 2000":
    GO TO 100
160 IF Monat<1 THEN GO TO 210
170 IF Monat>12 THEN GO TO 210
180 IF Jahr/4-INT(Jahr/4)=0 THEN LET m(2)=29: REM Schaltjahr
190 IF Tag>m(Monat) THEN PRINT "Dieser Monat hat nur  ";m;" Tage  ":
    GO TO 500
200 IF Tag>0 THEN GO TO 300
210 PRINT "So ein Unsinn. Gib mir reale Daten."
220 GO TO 500
300 REM Wandelt Datum in Anzahl der Tage seit Jahrhundertbeginn an um
310 LET j=Jahr-1901
320 LET b=365*j+INT (j/4): REM Anzahl der Tage zu Jahresbeginn
330 FOR n=1 TO (Monat-1: REM addiere zu den vergangenen Monaten
340 LET b=b+m(n): NEXT n
350 LET b=b+Tag
400 REM Wandelt in Wochentage um
410 LET b=b-7* INT (b/7)+1
420 PRINT Tag; "/"; Monat; "/"; Jahr
430 FOR n=6 TO 3 STEP -1: REM remove trailing spaces
440 IF d$(b,n) <> " " THEN GO TO 460
450 NEXT n
460 LET e$=d$
470 PRINT" ist ein ";e$
500 LET m(2)=28: REM Setzt den Februar um
510 INPUT "Nochmal?";a$
520 IF a$="n" THEN GO TO 540
```

```
  530 IF a$ <> "N" THEN GO TO 100
1000 REM Wochentage
1010 DATA "Montag", "Dienstag", "Mittwoch"
1020 DATA "Donnerstag", "Freitag", "Samstag", "Sonntag"
1100 REM Länge der Monate
1110 DATA 31, 28, 31, 30, 31, 30
1120 DATA 31, 31, 30, 31, 30, 31
```

Das erste Programm rechnet Fuß und Zoll in Yards um. Zur Erinnerung: Ein Yard (fast ein Meter) hat 3 Fuß, ein Fuß 12 Zoll.

```
   10 INPUT "yards?" ,yd, "feet?" ,ft, "inches?" ,in
   40 GO SUB 2000: REM Printe Werte aus
   50 PRINT"' = ";
   70 GO SUB 1000: REM Die Umrechnung
   80 GO SUB 2000: REM Drucke umgerechnete Werte aus
   90 PRINT
  100 GO TO 10
1000 REM Unterprogramm zur Umrechnung der yd, ft, in in die normale Form
       von yards, inches und feet
1010 LET in=36*yd+12*ft+in: REM Alles wurde in Inches umgerechnet
1030 LET s=SGN in: LET in=ABS in: REM Wir arbeiten im Positiv, gezeigt
       in s
1060 LET ft=INT (in/12): LET in=(in-12*ft)*s: REM Jetzt ist in umgerechnet
1080 LET yd=INT (ft/3)*s: LET ft=ft*s-3*yd: RETURN
2000 REM Unterprogramm zum printen von yd, ft und in
2010 PRINT yd;"yd";ft;"ft";in;"in";: RETURN
```

Hier ein Programm, mit dem man Münzen für das 'I Ging' werfen kann, das chinesische Buch der Wandlungen. Die Muster stehen zwar verkehrt herum, aber das macht Ihnen sicher nichts aus.

```
   5 RANDOMIZE
  10 FOR m=1 TO 6: REM Für 6 Würfe
  20 LET c=0: REM Setzt den Wert der Münze auf 0
  30 FOR n=1 TO 3: REM Für 3 Münzen
  40 LET c=c+2+INT (2*RND)
  50 NEXT n
  60 PRINT " ";
  70 FOR n=1 TO 2: REM 1. für das geworfene Hexagram, 2. für die
       Veränderungen
  80 PRINT "——";
  90 IF c=7 THEN PRINT "–";
 100 IF c=8 THEN PRINT " ";
 110 IF c=6 THEN PRINT "X";: LET c=7
 120 IF c=9 THEN PRINT "0";: LET c=8
 130 PRINT "—— ";
 140 NEXT n
 150 PRINT
 160 INPUT a$
 170 NEXT m: NEW
```

Geben Sie das Programm ein und lassen Sie es laufen. Danach müssen Sie fünfmal **ENTER** drücken, um die beiden Hexagramme zu erhalten. Diese Muster können Sie im Buch 'I Ging' nachschlagen. Der Text beschreibt eine Situation und was man tun kann oder soll. Wenn Sie angestrengt nachdenken, finden Sie einen Bezug zu Ihrem Leben. Wenn Sie das sechstemal **ENTER** drücken, löscht sich das Programm automatisch, damit man die Sache auch ernstnimmt.

Viele Leute finden die Texte oft viel passender, als der Zufall das nach ihrer Meinung erwarten ließe; ob das bei Ihrem ZX Spectrum auch der Fall ist, müssen Sie selbst entscheiden. Im allgemeinen sind Computer recht gottlose Gesellen.

Weiter geht es mit einem Programm, mit dem man 'Pangolin' spielen kann. Der Pangolin ist ein javanisches Schuppentier. Sie denken sich ein Tier aus, und der Computer versucht es zu erraten. Er stellt Ihnen dazu Fragen, auf die man mit 'yes' oder 'no' antworten kann. Wenn er von Ihrem Tier noch nie etwas gehört hat, verlangt er eine Frage, mit der er beim nächstenmal herausfinden kann, ob jemand ihm Ihr neues Tier eingegeben hat.

```
  5 REM pangolins
 10 LET nq=100: REM Anzahl der Fragen und Tiere
 15 DIM q$(nq,50): DIM a(nq,2): DIM r$(1)
 20 LET qf=8
 30 FOR n=1 TO qf/2-1
 40 READ q$(n): READ a(n,1): READ a(n,2)
 50 NEXT n
 60 FOR n=n TO qf-1
 70 READ q$(n): NEXT n
100 REM Spielbeginn
110 PRINT "Denken Sie sich ein Tier aus". "und drücken Sie dann eine
    Taste."
120 PAUSE 0
130 LET c=1: REM Beginn mit der ersten Frage
140 IF a(c,1)=0 THEN GO TO 300
150 LET p$=q$(c): GO SUB 910
160 PRINT "?": GO SUB 1000
170 LET in=1: IF r$="y" THEN GO TO 210
180 IF r$="Y" THEN GO TO 210
190 LET in=2: IF r$="n" THEN GO TO 210
200 IF r$<>"N" THEN GO TO 150
210 LET c=a(c,in): GO TO 140
300 REM Tier
310 PRINT "Meinen Sie vielleicht"
320 LET p$=q$(c): GO SUB 900: PRINT "?"
330 GO SUB 1000
340 IF r$="y" THEN GO TO 400
350 IF r$="Y" THEN GO TO 400
360 IF r$="n" THEN GO TO 500
370 IF r$="N" THEN GO TO 500
```

```
380 PRINT "Antworte mir", "wenn ich mit Dir rede!":
    GO TO 300
400 REM Erraten
410 PRINT "Das hatte ich mir doch gleich gedacht.": GO TO 800
500 REM Neues Tier
510 IF qf>nq-1 THEN PRINT "Ich bin sicher, Ihr Tier ist sehr interessant",
    "aber ich habe jetzt", "keine Zeit für sowas.": GO TO 800
520 LET q$(qf)=q$(c): REM Löschen des alten Tieres
530 PRINT "Was war es denn?": INPUT q$(qf+1)
540 PRINT "Stellen Sie mir eine Frage", "welcher Unterschied zwischen"
550 LET p$=q$(qf): GO SUB 900: PRINT "und einem ... besteht."
560 LET p$=q$(qf+1): GO SUB 900: PRINT " "
570 INPUT s$: LET b=LEN s$
580 IF s$(b)="?" THEN LET b=b-1
590 LET q$(c)=s$(TO b): REM Einführen der Frage
600 PRINT "Wie lautet die Antwort für"
610 LET p$=q$(qf+1): GO SUB 900: PRINT "?"
620 GO SUB 1000
630 LET in=1: LET io=2: REM Fragen nach neuen und alten Tieren
640 IF r$="y" THEN GO TO 700
650 IF r$="Y" THEN GO TO 700
660 LET in=2: LET io=1
670 IF r$="n" THEN GO TO 700
680 IF r$="N" THEN GO TO 700
690 PRINT "Das ist nicht gut." GO TO 600
700 REM Die Antworten erneuern
710 LET a(c,in)=qf+1: LET a(c,io)=qf
720 LET qf=qf+2: REM Jetzt den Platz für Tiere freimachen
730 PRINT "Das hat mich verwirrt."
800 REM Nochmal?
810 PRINT "Wollen Sie noch einen Versuch?": GO SUB 1000
820 IF r$="y" THEN GO TO 100
830 IF r$="Y" THEN GO TO 100
840 STOP
900 REM print without trailing spaces
905 PRINT " ";
910 FOR n=50 TO 1 STEP -1
920 IF p$(n)<>" " THEN GO TO 940
930 NEXT n
940 PRINT p$(TO n);: RETURN
1000 REM Antwort kriegen
1010 INPUT r$: IF r$="" THEN RETURN
1020 LET r$=r$(1): RETURN
2000 REM initial animals
2010 DATA "Lebt es im Wasser?" ,4,2
```

```
2020 DATA "Ist es schuppig?" ,3,5
2030 DATA "Ißt es Ameisen?" ,6,7
2040 DATA "einen Wal", "einen Mandelpudding", "ein Pangolin",
     "eine Ameise"
```

Dann ein Programm, das die britische Nationalflagge, den Union Jack, zeichnet.

```
   5 REM Union Jack
  10 LET r=2: LET w=7: LET b=1
  20 BORDER 0: PAPER b: INK w: CLS
  30 REM Schwarzen Hintergrund
  40 INVERSE 1
  50 FOR n=40 TO 0 STEP -8
  60 PLOT PAPER 0;7,n: DRAW PAPER 0;241,0
  70 NEXT n: INVERSE 0
 100 REM Weiße Teile zeichnen
 105 REM St. George
 110 FOR n=0 TO 7
 120 PLOT 104+n,175: DRAW 0,-35
 130 PLOT 151-n,175: DRAW 0,-35
 140 PLOT 151-n,48: DRAW 0,35
 150 PLOT 104+n,48: DRAW 0,35
 160 NEXT n
 200 FOR n=0 TO 11
 210 PLOT 0,139-n: DRAW 111,0
 220 PLOT 255,139-n: DRAW -111,0
 230 PLOT 255,84+n: DRAW -111,0
 240 PLOT 0,84+n: DRAW 111,0
 250 NEXT n
 300 REM St. Andrew
 310 FOR n=0 TO 35
 320 PLOT 1+2*n,175-n: DRAW 32,0
 330 PLOT 224-2*n,175-n: DRAW 16,0
 340 PLOT 254-2*n,48+n: DRAW -32,0
 350 PLOT 17+2*n,48+n: DRAW 16,0
 360 NEXT n
 370 FOR n=0 TO 19
 380 PLOT 185+2*n,140+n: DRAW 32,0
 390 PLOT 200+2*n,83-n: DRAW 16,0
 400 PLOT 39-2*n,83-n: DRAW 32,0
 410 PLOT 54-2*n, 140+n: DRAW -16,0
 420 NEXT n
 425 REM Einfüllen der Extrateile
 430 FOR n=0 TO 15
 440 PLOT 255,160+n: DRAW 2*n-30,0
 450 PLOT 0,63-n: DRAW 31-2*n,0
 460 NEXT n
```

```
470 FOR n=0 TO 7
480 PLOT 0,160+n: DRAW 14-2*n,0
485 PLOT 255,63-n: DRAW 2*n-,15,0
490 NEXT n
500 REM Rote Streifen
510 INVERSE 1
520 REM St. George
530 FOR n=96 TO 120 STEP 8
540 PLOT PAPER r;7,n: DRAW PAPER r;241,0
550 NEXT n
560 FOR n=112 TO 136 STEP 8
570 PLOT PAPER r;n,168: DRAW PAPER r;0,-113
580 NEXT n
600 REM St. Patrick
610 PLOT PAPER r;170,140: DRAW PAPER r;70,35
620 PLOT PAPER r;179,140: DRAW PAPER r;70,35
630 PLOT PAPER r;199,83: DRAW PAPER r;56,-28
640 PLOT PAPER r;184,83: DRAW PAPER r;70,-35
650 PLOT PAPER r;86,83: DRAW PAPER r;-70,-35
660 PLOT PAPER r;72,83: DRAW PAPER r;-70,-35
670 PLOT PAPER r;56,140: DRAW PAPER r;-56,28
680 PLOT PAPER r; 71,140: DRAW PAPER r;-70,35
690 INVERSE 0: PAPER 0: INK 7
```

Sie können auch versuchen, Ihre eigene Nationalflagge zu zeichnen. Trikoloren (auch die deutsche Fahne ist eine) sind ziemlich leicht, auch wenn manche Farben (das deutsche Gold) etwas schwieriger sind. Wenn Sie die Stars und Stripes zeichnen möchten, könnten Sie das Zeichen * mitverwenden.

Und zum Schluß ein Programm zum 'Hangman'-Spielen, also für das 'Henkerspiel'. Ein Spieler gibt ein Wort ein, der andere versucht es zu erraten. Das geht so:

```
  5 REM Henker
 10 REM Bildschirm aufbauen
 20 INK 0: PAPER 7: CLS
 30 LET x=240: GO SUB 1000: REM Männchen zeichnen
 40 PLOT 238,128: DRAW 4,0: REM Mund
100 REM Wortaufbau
110 INPUT w$: REM Das zu erratene Wort
120 LET b=LEN w$: LET v$=" "
130 FOR n=2 TO b: LET v$=V$+" "
140 NEXT n: REM v$=Buchstaben, die bisher erraten wurden
150 LET c=0: LET d=0: REM Rate- und Fehlerzähler
160 FOR n=0 TO b-1
170 PRINT AT 20,n;"-";
180 NEXT n: REM Schreibe -'s statt Buchstaben
200 INPUT "Rate einen Buchstaben:  ";g$
```

```
210 IF g$="" THEN GO TO 200
220 LET g$=g$(1): REM Nur der erste Buchstabe
230 PRINT AT 0,c;g$
240 LET c=c+1: LET u$=v$
250 FOR n=1 TO b: REM Wort, soweit es geraten wurde
260 IF w$(n)=g$ THEN LET v$(n)=g$
270 NEXT n
280 PRINT AT 19,0;v$
290 IF v$=w$ THEN GO TO 500: REM Wort erraten
300 IF v$<>u$ THEN GO TO 200: REM Richtig geraten
400 REM Zeichne nächsten Teil des Galgen
410 IF d=8 THEN GO TO 600: REM Gehängt
420 LET d=d+1
430 READ x0,y0,x,y
440 PLOT x0,y0: DRAW x,y
450 GO TO 200
500 REM Freier Mann
510 OVER 1: REM Mann löschen
520 LET x=240: GO SUB 1000
530 PLOT 238,128: DRAW 4,0: REM Mund
540 OVER 0: REM Das Männchen wieder zeichnen
550 LET x=146: GO SUB 1000
560 PLOT 143,129: DRAW 6,0, PI/2: REM Grinsen
570 GO TO 800
600 REM Mann hängen
610 OVER 1: REM Boden löschen
620 PLOT 255,65: DRAW -48,0
630 DRAW 0,-48: REM Falltür öffnen
640 PLOT 238,128: DRAW 4,0: REM Mund löschen
650 REM Glieder bewegen
655 REM Arme
660 PLOT 255,117: DRAW -15,-15: DRAW -15,15
670 OVER 0
680 PLOT 236,81: DRAW 4,21: DRAW 4,-21
690 OVER 1: REM Beine
700 PLOT 255,66: DRAW -15,15: DRAW -15,-15
710 OVER 0
720 PLOT 236,60: DRAW 4,21: DRAW 4,-21
730 PLOT 237,127: DRAW 6,0, -PI/2: REM Finster dreinschauen
740 PRINT AT 19,0;w$
800 INPUT "Nochmal";a$
810 IF a$="" THEN GO TO 850
820 LET a$=a$(1)
830 IF a$="n" THEN STOP
840 IF a$(1)="N" THEN STOP
850 RESTORE: GO TO 5
1000 REM Zeichne Mann ab Spalte x
```

```
1010 REM Kopf
1020 CIRCLE x,132,8
1030 PLOT x+4,134: PLOT x-4,134: PLOT x,131
1040 REM Körper
1050 PLOT x,123: DRAW 0,-20
1055 PLOT x,101: DRAW 0,-19
1060 REM Beine
1070 PLOT x-15,66: DRAW 15,15: DRAW 15,-15
1080 REM Arme
1090 PLOT x-15,117: DRAW 15,-15: DRAW 15,15
1100 RETURN
2000 DATA 120,65,135,0,184,65,0,91
2010 DATA 168,65,16,16,184,81,16,-16
2020 DATA 184,156,68,0,184,140,16,16
2030 DATA 204, 156,-20,-20,240,156,0,-16
```

Binär und hexadezimal

Dieser Anhang beschreibt, wie Computer mit Hilfe des Binärsystems zählen.

Die meisten europäischen Sprachen zählen nach einer mehr oder weniger regelmäßigen Zehnermethode, die nach einem nicht ganz so regelmäßigen Anfang zu ganz regelmäßigen Gruppen führt:

zwanzig, einundzwanzig, zweiundzwanzig,..., neunundzwanzig
dreißig, einunddreißig, zweiunddreißig,..., neununddreißig
vierzig, einundvierzig, zweiundvierzig,..., neunundvierzig

und so weiter. Noch systematischer wird das durch die arabischen Zahlen, die wir verwenden. Der einzige Grund dafür, warum wir die Zehn benutzen, ist der, daß wir zehn Finger mit Daumen haben.

Statt das *Dezimal*system zu benutzen, das Zehn als Basis hat, verwenden Computer eine binäre Form, die *Hexadezimal*system (kurz Hex) genannt wird. Es beruht auf der Sechzehn. Da in unserem Zahlensystem nur zehn Ziffern zur Verfügung stehen, brauchen wir sechs zusätzliche Ziffern zum Zählen. Wir verwenden dazu A, B, C, D, E und F. Und was kommt nach F? So, wie wir mit zehn Fingern für zehn 10 schreiben, schreiben die Computer 10 für sechzehn. Ihr Zahlensystem beginnt mit:

Hex	*Deutsch*
0	null
1	eins
2	zwei
:	:
:	:
9	neun

genau wie bei unserer Methode, aber dann geht es weiter

A	zehn
B	elf
C	zwölf
D	dreizehn
E	vierzehn
F	fünfzehn
10	sechzehn
11	siebzehn
:	:
:	:
19	fünfundzwanzig
1A	sechsundzwanzig
1B	siebenundzwanzig
:	:
:	:
1F	einunddreißig
20	zweiunddreißig

21	dreiunddreißig
:	:
:	:
9E	hundertachtundfünfzig
9F	hundertneunundfünfzig
A0	hundertsechzig
A1	hunderteinundsechzig
:	:
B4	hundertachtzig
:	:
FE	zweihundertvierundfünfzig
FF	zweihundertfünfundfünfzig
100	zweihundertsechsundfünfzig

Wenn Sie Hexschreibung verwenden und das ganz deutlich machen wollen, fügen Sie am Ende der Zahl 'h' an und sprechen das 'hex' aus. Beispiel: Für hundertachtundfünfzig schreiben Sie '9Eh' und sagen 'neun E hex'.

Sie werden sich fragen, was das alles mit Computern zu tun hat. In der Tat verhalten Computer sich so, als hätten sie nur zwei Ziffern, dargestellt durch geringen Stromdurchgang oder aus (0) und hohen Stromdurchgang oder ein (1). Das nennt man *Binärsystem*, und die beiden Binärziffern heißen *Bits*. Ein Bit ist also entweder 0 oder 1.

Bei den verschiedenen Systemen beginnt das Zählen so:

deutsch	*dezimal*	*hexadezimal*	*binär*
null	0	0	0 oder 0000
eins	1	1	1 oder 0001
zwei	2	2	10 oder 0010
drei	3	3	11 oder 0011
vier	4	4	100 oder 0100
fünf	5	5	101 oder 0101
sechs	6	6	110 oder 0110
sieben	7	7	111 oder 0111
acht	8	8	1000
neun	9	9	1001
zehn	10	A	1010
elf	11	B	1011
zwölf	12	C	1100
dreizehn	13	D	1101
vierzehn	14	E	1110
fünfzehn	15	F	1111
sechzehn	16	10	10000

Der wichtigste Punkt: Sechzehn entspricht zwei hoch vier, und dadurch wird die Umrechnung zwischen hex und binär sehr leicht.

Zur Umwandlung von Hex- in Binärzahlen verwandeln Sie jede Hexziffer mit Hilfe der obigen Tabelle in vier Bits.

Bei der Umwandlung von Binär- in Hexzahlen teilen Sie die Binärzahl in Gruppen von vier Bits auf, wobei Sie rechts beginnen, und verwandeln dann jede Gruppe in die entsprechende Hexziffer.

Aus diesem Grund schreiben die Menschen, obwohl streng genommen Computer ein rein binäres System verwenden, die in einem Computer gespeicherten Zahlen oft in der Hex-Schreibweise.

Die Bits im Inneren des Computers sind meistens in Sätzen von acht gruppiert, den sogenannten *Bytes.* Ein einzelnes Byte kann jede Zahl von Null bis zweihundertfünfundfünfzig darstellen (11111111 binär oder FF hex) oder alternativ jedes Zeichen im Zeichenvorrat des ZX Spectrum. Sein Wert kann mit zwei Hexziffern geschrieben werden.

Zwei Bytes können gruppiert werden zu einem sogenannten Wort. Ein Wort kann mit sechzehn Bits oder vier Hexziffern geschrieben werden und stellt eine Zahl von Ø bis (dezimal) $2^{16}-1$ dar = 65535.

Ein Byte besteht immer aus acht Bits, aber Wörter unterscheiden sich von Computer zu Computer durch ihre Länge.

Die **BIN**-Schreibweise in Kapitel 14 liefert ein Mittel, auf dem ZX Spectrum Zahlen binär zu schreiben: '**BIN** Ø' steht für null, '**BIN** 1' für eins, '**BIN** 1Ø' für zwei, und so weiter.

Dafür können Sie nur Ø und 1 verwenden, also muß die Zahl eine nicht negative ganze Zahl sein, beispielsweise können Sie nicht '**BIN**–11' für minus drei schreiben. Sie müssen stattdessen schreiben '–**BIN** 11'. Die Zahl darf nicht größer sein als 65535 dezimal, das heißt, sie kann nicht mehr haben als sechzehn Bits.

ATTR war in Wirklichkeit binär. Wenn Sie das Resultat von **ATTR** in die binäre Schreibweise übertragen, können Sie es in acht Bits schreiben.
Das erste ist 1 für Blinken, Ø für gleichbleibend.
Das zweite ist 1 für hell, Ø für normal.
Die nächsten drei sind der Code für die Papierfarbe, binär geschrieben.
Die letzten drei sind der Code für die Inkfarbe, binär geschrieben.

Die Farbencodes verwenden ebenfalls die binäre Schreibweise: Jeder binär geschriebene Code kann in drei Bits geschrieben werden, der erste für grün, der zweite für rot und der dritte für blau.

Schwarz hat überhaupt kein Licht, also sind alle Bits Ø (aus). Der Code für Schwarz ist daher binär ØØØ oder Null.

Die reinen Farben Grün, Rot und Blau haben von den drei Bits nur ein Bit 1. Ihre Binärcodes sind 1ØØ, Ø1Ø und ØØ1 oder vier, zwei und eins.

Die anderen Farben sind Gemische davon, also haben ihre Binärcodes zwei oder mehr 1-Bits.

Register

Das Register enthält auch die Tasten der Tastatur und wie man sie erreicht (Modus **K**, **L**, **E**, **C** oder **G** und welche Umschalttaste – Shift – wo richtig ist).

In der Regel wird ein Eintrag pro Kapitel nur einmal erwähnt. Wenn Sie einen Hinweis gefunden haben, lesen Sie deshalb am besten das Kapitel samt den Übungen durch.

)	K, L oder C, SYMBOL SHIFT 9.
*	K, L oder C, SYMBOL SHIFT B.
+	K, L oder C, SYMBOL SHIFT K.
,	K, L oder C, SYMBOL SHIFT H.
–	K, L oder C, SYMBOL SHIFT J.
.	K, L oder C, SYMBOL SHIFT M.
/	K, L oder C, SYMBOL SHIFT V.
:	K, L oder C, SYMBOL SHIFT Z.
;	K, L oder C, SYMBOL SHIFT O.
<	K, L oder C, SYMBOL SHIFT R.
=	K, L oder C, SYMBOL SHIFT L
>	K, L oder C, SYMBOL SHIFT T.
?	K, L oder C, SYMBOL SHIFT C.
@	K, L oder C, SYMBOL SHIFT 2.
[	E, Shift Y.
\	E, Shift D.
]	E, Shift U.
↑	K, L oder C, SYMBOL SHIFT H.
_	K, L oder C, SYMBOL SHIFT Ø.
£	K, L oder C, SYMBOL SHIFT X.
{	E, Shift F.
|	E, Shift S.
}	E, Shift G.
~	E, Shift A.
©	E, Shift P.
<=	K, L oder C, SYMBOL SHIFT Q.
>=	K, L oder C, SYMBOL SHIFT E.
<>	K, L oder C, SYMBOL SHIFT W.
◖	K, L oder C, CAPS SHIFT 5.
◗	K, L oder C, CAPS SHIFT 8.
▬	K, L oder C, CAPS SHIFT 6.
▲	K, L oder C, CAPS SHIFT 7.

Notizen:

Notizen:

Verlag Cooperation, München
ISBN 3-88945-011-3

ZX SPECTRUM BASIC-