Debugging — Fehlersuche in Basic-Programmen

Diese Situation kennt wohl jeder Besitzer eines Homecomputers zur Genüge: Da tippt man in stundenlanger Arbeit ein ellenlanges Listing ein, lehnt sich nach dem letzten »RETURN« einen Augenblick erleichtert zurück, gibt das magische Wort »RUN« ein — und natürlich läuft das Programm in keiner Weise so, wie es eigentlich sollte. Das Spektrum der möglichen Ereignisse reicht dabei vom simplen »SYNTAX ERROR« bis zum völligen Absturz des Programms.

Solange der Computer noch brav seine Fehlermeldungen ausgibt, hat man ja noch Glück gehabt. Kritisch wird die Situation dann, wenn auf dem Bildschirm ein eigenartiges Gemisch undefinierbarer Zeichen erscheint und sich der Computer weder durch Betätigen aller erreichbaren Tasten, noch durch gutes Zureden wieder auf den Boden der Tatsachen zurückholen läßt. Wenn man beim Eintippen eines Programms einmal an diesem Punkt angelangt ist, wird es Zeit, sich die erste Regel gut einzuprägen: Jedes Programm sollte vor dem Start unbedingt mit SAVE gesichert werden.

Mit dem SAVEn allein ist es allerdings noch nicht getan, es muß effektiv noch etwas gegen die im Programm enthaltenen Fehler unternommen werden. Diesen Vorgang bezeichnet man auch als »Debugging«. Das Wort ist von der englischen Bezeichnung »Bug« abgeleitet und bedeutet eigentlich »entwanzen«, wobei mit den »Wanzen« die Fehler gemeint sind, die sich überall im Programm verstecken. In der amerikanischen Umgangssprache hat sich das Wort ganz allgemein für das Suchen versteckter Fehler eingebürgert.

Ganz grob kann man zwischen zwei Arten von Fehlern unterscheiden. Einerseits gibt es die logischen Fehler, die mit schöner Regelmäßigkeit in der Entwicklungsphase eines Programms auftauchen, weil man dem Computer noch nicht genau genug gesagt hat, was er denn nun eigentlich machen soll. Diese Art von Fehlern erkennt man zumeist daran, daß das Programm anstandslos läuft, aber nicht immer die gewünschten Ergebnisse produziert. Die zweite Art von Fehlern ist wesentlich profanerer Natur und tritt praktisch jedesmal dann auf, wenn man ein Programm von einem fremden Listing oder auch von den eigenen Aufzeichnungen abtippt: Es handelt sich dann zumeist um schlichte Tippfehler oder um Fehler, die auf schlechter Lesbarkeit der Vorlage beruhen. Wir wollen uns im folgenden nur mit der zweiten Art von Fehlern beschäftigen.

Der Computer hilft bei der Fehlersuche

Wie geht man nun zweckmäßig vor, um alle Fehler zu finden, ohne das gesamte Programm von Anfang bis Ende mit der Vorlage vergleichen zu müssen? Nun, eine allgemeingültige Methode, die für alle Arten von Programmen anwendbar wäre, gibt es leider nicht. Dennoch erscheint ein gewisses systematisches Vorgehen durchaus angebracht.

Zunächst sollten wir uns darüber klarwerden, inwieweit uns der Computer selbst bei der Suche nach Fehlern helfen kann. Als erstes kommen einem dabei natürlich die Fehlermeldungen in den Sinn, die beim Commodore-Basic ja erfreulicherweise im Klartext erfolgen und recht vielfältig sind. Wer mit den englischen Bezeichnungen nicht sofort etwas anfangen kann, hat die Möglichkeit, die deutschen Erläuterungen dazu im Handbuch nachzuschlagen.

Was aber soll man tun, wenn der Computer gar keine Fehlermeldung ausgibt, sondern sich nach »RUN« einfach sang- und klanglos verabschiedet und auf keine Tasten mehr reagiert?

Nun, für solche Fälle gibt es im Basic des C 64 beziehungsweise des VC 20 zwei spezielle Befehle, die man immer dann in nicht zu geringem Umfang einsetzen sollte, wenn man nicht genau weiß, wo denn nun der Fehler steckt. Gemeint sind die Basic-Befehle STOP und CONT. Wenn der Computer beim Abarbeiten des Programms auf den Befehl STOP stößt, unterbricht er die Programmausführung und gibt eine Meldung »BREAK IN nnn« aus, wobei nnn die Zeilennummer ist, in der er die STOP-Anweisung gefunden hat. Alle Variablen und auch der Stackpointer bleiben dabei erhalten, so daß die STOP-Anweisung auch in Unterprogrammen und innerhalb von FOR-NEXT-Schleifen auftreten kann.

Nach einem solchermaßen erzwungenen Programmstop kann man sich im Direktmodus mit dem PRINT-Befehl über die Werte wichtiger Variablen informieren und sogar mit LIST einzelne Programmteile anschauen. Danach gibt man den CONT-Befehl, und das Programm wird ganz normal fortgesetzt. Wenn es zu Testzwecken notwendig erscheint, kann man während eines Stops auch im Direktmodus Varia-

blenwerte verändern oder FOR-NEXT-Schleifen verwenden. Allerdings dürfen weder neue Programmzeilen eingegeben noch alte gelöscht oder verändert werden, da dadurch gleichzeitig alle Variablen gelöscht werden und CONT danach nicht mehr möglich ist.

Es ist empfehlenswert, an kritischen Stellen im Programm STOP-Befehle einzufügen, um dadurch den Programmlauf verfolgen zu können und den Fehler immer mehr einzugrenzen. Kritische Stellen sind generell und ohne Ausnahme alle SYS- und USR-Aufrufe, desgleichen alle POKE-Befehle, über deren Bedeutung man sich nicht hundertprozentig im klaren ist. Im Zweifelsfalle sollte man auch nach jedem GOSUB im Programm zunächst einen STOP-Befehl einbauen, um sicherzugehen, daß das Unterprogramm auch wieder auf normalem Wege verlassen wird.

Fehlersuche mit STOP und PRINT

Jedesmal, wenn man die Harmlosigkeit, zum Beispiel eines SYS-Befehls, durch davor und danach plazierte STOP-Befehle festgestellt hat, kann man die STOPs natürlich wieder entfernen, um einen flüssigeren Programmablauf zu erreichen. Es empfiehlt sich, alle eingefügten STOP-Befehle auf einem Zettel zu notieren, um die Übersicht zu behalten. Schließlich dienen diese Befehle nur der Fehlersuche und müssen irgendwann einmal alle wieder entfernt werden.

In vielen Fällen kann man STOP-Befehle durch einfache PRINT-Anweisungen ersetzen. Das hat den Vorteil, daß keine Programmunterbrechung stattfindet und man nicht jedesmal CONT eintippen muß. Außerdem kann man in PRINT-Anweisungen auch zusätzliche Informationen geben, zum Beispiel Variablenwerte ausdrucken oder direkt auf ein spezielles Problem aufmerksam machen. Diese PRINT-Anweisungen sollten aber in irgendeiner Weise von den normalen Bildschirmausgaben unterschieden sein. Zum Beispiel kann man jede PRINT-Anweisung zur Fehlersuche mit fünf Sternchen oder fünf Pluszeichen beginnen lassen.

Will man sich mehrere Variable während des Programmlaufs ausdrucken lassen, sind kleine Unterprogramme recht hilfreich, die in einen freien Zeilenbereich geschrieben werden und die alle benötigten Ausgaben durchführen. Am Ende solcher Unterprogramme sollte eine GET-Schleife stehen, die das Programm auf Tastendruck weiterlaufen läßt. Statt langer PRINT-Listen braucht man so nur einen GOSUB-Aufruf überall dort im Programm einzufügen, wo dies sinnvoll erscheint.

Komfortables Debugging mit Basic-Erweiterungen

Das Arbeiten mit STOP und CONT mag manchem Computerneuling etwas ungewohnt erscheinen, aber es ist jedenfalls ein recht sicheres Mittel, einem immer wieder abstürzenden Programm auf die Schliche zu kommen.

Leider sind STOP und CONT auch schon die einzigen speziellen Debug-Befehle im Commodore-Basic. Viele Spracherweiterungen, wie zum Beispiel das bekannte Simons Basic oder Exbasic Level II, stellen zusätzliche Funktionen zur schnellen Fehlersuche zur Verfügung. Wichtige derartige Befehle sind zum Beispiel

* **TRACE** — listet während des Programmlaufs die gerade bearbeiteten Programmzeilen oder zumindest die Zeilennummern auf.

* **DUMP** — gibt eine Liste aller Variablen mit ihren derzeitigen Werten aus.

* **FIND** — sucht im Direktmodus eine Zeichenfolge im Programm.

* **ON ERROR GOTO ...** — ermöglicht die Fehlerbehandlung im Programm selbst (ohne Abbruch).

Wenn der Computer jedoch noch in manierlicher Weise seine Fehlermeldungen ausgibt, sind aufwendige Verfahren zumeist nicht nötig, denn zusammen mit der Meldung über die Art des Fehlers erfährt man ja auch die Zeilennummer des Auftretens.

Syntax Error: Wenn der Computer nur noch »Bahnhof« versteht

Die häufigste Fehlermeldung ist sicherlich der ungeliebte »SYNTAX ERROR«. Böse Zungen behaupten allerdings, beim VC 20 wäre es der »OUT OF MEMORY ERROR«. Wie dem auch sei, solange der Computer nur Syntax-Fehler meldet, kann man noch von Glück reden. Es handelt sich dabei meistens um einfache Tippfehler, die nach auflisten der entsprechenden Zeile leicht zu finden und zu korrigieren sind. Beliebte Fehler sind zum Beispiel fehlende oder überzählige Klammern und die Verwechselung ähnlicher Zeichen, wie zum Beispiel »0« und »O«, »1« und »I« oder »8« und »B«. Sehr häufig ist auch die Verwechslung von Punkt und Komma, was sich besonders in DATA-Zeilen verhängnisvoll auswirken kann, wie wir nachher noch sehen werden. Wenn Sie also irgendwo einen »SYNTAX ERROR« gemeldet bekommen und in der fraglichen Zeile auf Anhieb keinen Fehler finden, dann gehen Sie zuerst die vorhin genannten Punkte durch.

Eine gute Hilfe ist es, mit dem Cursor die fehlerhafte Zeile Zeichen für Zeichen abzufahren und dabei mit der Vorlage zu vergleichen. Kommen in der Fehlerzeile viele Klammern vor, dann empfiehlt sich häufig das Anlegen zweier Strichlisten für öffnende und schließende Klammern. Die Anzahlen müssen innerhalb jeder Basic-Anweisung übereinstimmen. Aber bitte keine Klammern mitzählen, die in Anführungszeichen stehen; diese haben mit der Syntax nichts zu tun.

Syntax-Fehler, die man nicht sehen kann

Ab und zu kann es vorkommen, daß man bei aller Sorgfalt einen Syntaxfehler nicht findet, wie zum Beispiel in der folgenden Basic-Zeile:

```
10 OPEN 1,4:PRINT#1,"HALLO":CLOSE 1
```

Wenn der Computer hier dennoch einen Syntaxfehler meldet, dann kann das nur eine Ursache haben: Bei der Eingabe dieser Zeile wurden die Basic-Befehle in der bekannten Art und Weise abgekürzt. Der zweite Befehl wurde also als »?#1« eingegeben, was beim Auflisten wieder zu »PRINT#1« wird. Leider sind PRINT und PRINT# zwei völlig verschiedene Befehle, genauso wie INPUT und INPUT# oder GET und GET#. Alle diese #-Befehle darf man daher nie abkürzen. Die normalen PRINT-, INPUT- und GET-Routinen wissen nämlich mit dem nachfolgenden »#«

nichts anzufangen und es kommt zu einer Fehlermeldung.

Eine ähnliche Situation kann sehr leicht bei Verwendung langer Variablennamen auftreten. In einem Spielprogramm »Schiffe versenken« kann zum Beispiel die folgende Zeile auftreten:

10 ANZAHLSCHIFFE = 12

In dieser Zeile wird unweigerlich ein »SYNTAX ERROR« auftreten, weil der Computer innerhalb des Variablennamens »ANZAHLSCHIFFE« das Basic-Schlüsselwort »IF« entdeckt und sich bei aller Anstrengung nicht erklären kann, was eine IF-Abfrage an dieser Stelle soll. Trotz aller Vorteile für die Übersichtlichkeit eines Programms sei daher an dieser Stelle von der Benutzung langer Variablennamen abgeraten.

Was tun bei »OUT OF DATA«?

Eine andere häufig auftretende Fehlermeldung ist vor allem bei Anfängern gefürchtet, nämlich der »OUT OF DATA ERROR«. Gefürchtet ist dieser Fehler vor allem deswegen, weil die Zeilennummer, die der Computer zu dieser Fehlermeldung ausgibt, in den allermeisten Fällen keinen Hinweis darauf gibt, an welcher Stelle denn nun ein Fehler vorliegt. Listet man nämlich die fehlerhafte Zeile am Bildschirm auf, so findet man dort nur den READ-Befehl, für den keine DATAs mehr vorhanden waren. In der Regel steht dieser READ-Befehl innerhalb einer FOR-NEXT-Schleife. Ein typisches, wenn auch stark vereinfachtes Beispiel für das Auftreten von Fehlern im Zusammenhang mit DATA-Anweisungen ist in Listing 1 auf Seite 50 gegeben. In den Zeilen 100 bis 170 wird eine Prüfsumme über den ersten DATA-Block gebildet, und nur dann, wenn diese Prüfsumme in Ordnung ist, wird in den nächsten Programmteil verzweigt, wo aus dem zweiten DATA-Block Zahlen gelesen und an den Anfang des Bildschirms gePOKEt werden und dort das Wort »COMMODORE« bilden sollen.

Das Programm enthält nun einige Fehler in den DATA-Zeilen, die wir gemeinsam herausfinden wollen. Stellen wir uns einfach vor, wir hätten das Programm in Listing 1 aus einer Zeitschrift abgetippt und dabei einige Fehler in den DATA-Zeilen fabriziert. In Listing 2 sind zum Vergleich noch einmal die entsprechenden DATA-Zeilen des »Original«-Listings abgedruckt. Die Fehlersuche scheint somit recht einfach: Man vergleicht die paar DATAs in beiden Listings und wird dann schon den Fehler finden. Bei diesem kurzen Testprogramm stimmt das natürlich auch. Aber stellen wir uns doch einmal vor, daß die beiden DATA-Blöcke insgesamt vielleicht über drei volle Listing-Seiten gehen und nicht nur über drei Zeilen wie in unserem Beispiel. Dann lohnt es sich nämlich mit Sicherheit schon, wenn man etwas systematischer an die Fehlersuche herangeht.

Zuerst starten wir unser Programm nach Listing 1 einmal ganz arglos mit RUN, nachdem wir es vorher auf Kassette oder Diskette abgespeichert hatten. Der Programmverlauf ist zu Anfang ganz wie erwartet: Der Bildschirm wird gelöscht, es erscheint die Meldung »S = 270« und darunter »OK«, dann jedoch erscheinen am oberen Bildschirmrand statt eines längeren Wortes nur die beiden Zeichen »@« und »C« und das Programm bricht mit der Meldung »?ILLEGAL QUANTITY ERROR IN 230« ab. Was ist hier geschehen?

Wenn wir uns Zeile 230 einmal auflisten lassen, dann sehen wir

230 POKE B+I,X

Da wir wissen, daß nur Zahlen zwischen 0 und 255 gePOKEt werden können, vermuten wir den Fehler beim Wert der Variablen X. Um unsere Vermutung zu bestätigen, fragen wir den Computer doch einmal ganz einfach nach dem Wert von X, indem wir eintippen

PRINT X

und danach die RETURN-Taste betätigen. Wir erhalten als Antwort den Wert 1513, der tatsächlich zu groß ist, um in eine Speicherzelle zu passen. Wir vergleichen den zweiten DATA-Block mit dem Original (Listing 2) und stellen fest, daß wir bei der Eingabe das Komma zwischen den beiden Zahlen 15 und 13 in Zeile 360 vergessen haben. Das ändern wir, indem wir das Komma nachträglich einfügen. Dank dieses schnellen Erfolges bessert sich unsere Stimmung um einiges, was sich jedoch nach dem nächsten RUN sehr schnell wieder ändert. Zwar erscheint am Bildschirm zunächst ganz ordentlich die Prüfsumme des ersten DATA-Blocks und das dazugehörige »OK«, aber am oberen Bildschirmrand stimmt einiges noch nicht: Man liest dort die Zeichenfolge »@COMMOORE« statt »COMMODORE«.

Auf den ersten Blick würde man vielleicht vermuten, daß der Fehler nur im zweiten DATA-Block stehen kann, weil das Lesen des ersten Blocks keine Fehlermeldung erzeugt und sogar die Prüfsummenbildung stimmt. Diese Überlegung ist aber nicht ganz schlüssig. Denn durch Bildung einer einfachen Prüfsumme werden Vertauschungsfehler und überflüssige oder fehlende Nullen nicht erkannt. Die fünf DATA-Zeilen in Listing 3 ergeben zum Beispiel alle die gleiche Prüfsumme.

Man sollte sich also nie blindlings auf Prüfsummen verlassen. Sie sind zwar oft nützlich, um Fehler in DATA-Zeilen festzustellen, man darf aber aus der Richtigkeit der Prüfsummenprobe niemals auf die Abwesenheit von Fehlern schließen. Außerdem taucht eine weitere Schwierigkeit auf: Wenn man nur eine globale Prüfsumme über alle DATAs bildet, dann kann man zwar unter Umständen einen Fehler nachweisen, weiß aber immer noch nicht, wo er steckt. Da muß man dann schon zu anderen Mitteln greifen.

Mit Listing, Lupe und Logik« Dem Fehler auf der Spur

Um den Fehler aufzuspüren, können wir dem Computer einen großen Teil der Arbeit überlassen. Als erstes wollen wir feststellen, in welchem der beiden DATA-Blöcke der Fehler liegen könnte. Dazu veranlassen wir den Computer einfach, nur den ersten DATA-Block zu lesen, indem wir eine STOP-Anweisung hinter die erste FOR-NEXT-Schleife plazieren. Wir fügen also folgende Zeile ins Programm ein:

145 STOP

Wenn wir das Programm jetzt starten, erhalten wir die Meldung »BREAK IN 145«, die von unserem STOP-Befehl herrührt. Da aber das Programm bis Zeile 145 durchlaufen wurde, muß der erste DATA-Block an dieser Stelle vollständig gelesen worden sein. Die Variable X enthält natürlich immer noch den zuletzt gelesenen DATA-Wert. Wenn dieser Programmteil richtig gearbeitet hat, dann müßte X jetzt den Wert Null haben, denn dies ist ja gerade der letzte DATA-Wert aus Block 1, wie man anhand von Listing 1 oder 2 unschwer erkennen kann. Das können wir einfach nachprüfen, indem wir den Computer nach dem Wert von X fragen:

PRINT X

Zu unserem Erstaunen ist die Antwort aber nicht 0, sondern 12. Wir

werfen wieder einen Blick auf Listing 1 und stellen fest, daß die Zahl 12 die vorletzte Zahl im ersten DATA-Block ist. Offenbar wurde eine Zahl zuwenig gelesen! Es ist nun verlockend, einfach den Endwert der ersten FOR-NEXT-Schleife um eins zu erhöhen, um alle Werte des ersten Blocks zu lesen. Doch halt, hier ist Vorsicht geboten. Viel wahrscheinlicher als ein Fehler in einer FOR-NEXT-Schleife ist ein Fehler innerhalb der DATA-Zeilen. Bei so vielen Zahleneingaben kann man sich schließlich leicht mal vertippen. Betrachten wir das Problem also einmal von der anderen Seite. Wenn die Anzahl der gelesenen X-Werte stimmt, das Programm aber trotzdem nur bis zum vorletzten DATA-Wert kommt, dann enthält Block 1 vielleicht einen DATA-Wert zuviel. Wir wollen also die DATAs in Block 1 ganz gezielt überprüfen. Dazu schreiben wir in einen freien Zeilenbereich, zum Beispiel ab Zeile 1000, das folgende kleine Unterprogramm:

```
1000 PRINT "I =" ; I , "X =" ; X
1010 GET A$ : IF A$ < > CHR$(32)
THEN 1010
1020 RETURN
```

In die erste FOR-NEXT-Schleife fügen wir direkt hinter die READ-Anweisung einen Aufruf dieses Unterprogramms ein:

```
125 GOSUB 1000
```

Wenn wir das Programm nun laufen lassen, geschieht folgendes: Der Computer gelangt mit Zeile 110 in die Leseschleife. In Zeile 120 wird jeweils ein DATA-Element gelesen. Dann erfolgt mit der eingefügten Zeile 125 ein Sprung in das vorhin geschriebene Unterprogramm. Dieses Unterprogramm druckt den Wert der Zählvariablen I und den soeben gelesenen DATA-Wert X aus und wartet dann, bis die Leertaste betätigt wird. Dann kehrt das Unterprogramm zurück und die Schleife wird nach dem NEXT in Zeile 140 erneut durchlaufen. Auf diese Art und Weise erhält man am Bildschirm eine übersichtliche Darstellung der gelesenen DATA-Werte, die man leicht mit der Vorlage vergleichen kann. Wenn wir das Programm jetzt starten, erhalten wir jeweils nach Drücken der Leertaste eine Bildschirmanzeige, etwa in der folgenden Art:

```
I =  1   X =  12
I =  2   X =  33
I =  3   X =  11
```

und so weiter bis schließlich das Ende von DATA-Zeile 310 erreicht wird:

```
I =  9   X =  18
I =  10  X =   0
```

Nanu? Das hatten wir eigentlich nicht erwartet. X = 18 ist der letzte Wert in Zeile 310, und danach sollte eigenlich der erste Wert aus der nächsten DATA-Zeile gelesen werden, nämlich X = 11. Woher also kommt dieser Wert Null bei I = 10? Ein Vergleich von Zeile 310 in Listing 1 (abgetippt) mit Listing 2 (Original) führt uns auf des Rätsels Lösung. Offenbar haben wir beim Abtippen am Ende von Zeile 310 noch ein Komma gesetzt, was da nicht hingehört. Ein Komma in einer DATA-Anweisung trennt für unseren Commodore-Computer aber immer zwei Werte voneinander, und da er hinter dem letzten Komma nichts mehr findet, setzt er kurzerhand den Wert Null dafür an.

Damit haben wir den überzähligen DATA-Wert im ersten Block gefunden. Wir entfernen das Komma in Zeile 310, löschen die Zeile 125 mit dem GOSUB-Befehl und ebenso die Zeile 145 mit dem nun nicht mehr benötigten STOP-Befehl.

Ein erneuter Probelauf des Programms schreibt die Zeichenfolge »COMMOORE« links oben in den Bildschirm — und bringt die Fehlermeldung »OUT OF DATA ERROR IN 220«. Nach Auflisten der Zeile 220 sehen wir leider nur

```
220 READ X
```

Das bringt uns nicht viel weiter. Die Fehlermeldung und das verstümmelte Wort »COMMOORE« am oberen Bildschirmrand deuten aber auf einen fehlenden DATA-Wert in Block 2 hin. Untersuchen wir also Block 2 einmal genauer. Das Unterprogramm zum Ausdrucken der Werte von I und X am Bildschirm befindet sich ja ab Zeile 1000 noch im Speicher. Wir brauchen daher nur einen entsprechenden GOSUB-Befehl in die zweite Leseschleife einzufügen, am besten gleich nach dem READ-Befehl, also etwa in Zeile 225:

```
225 GOSUB 1000
```

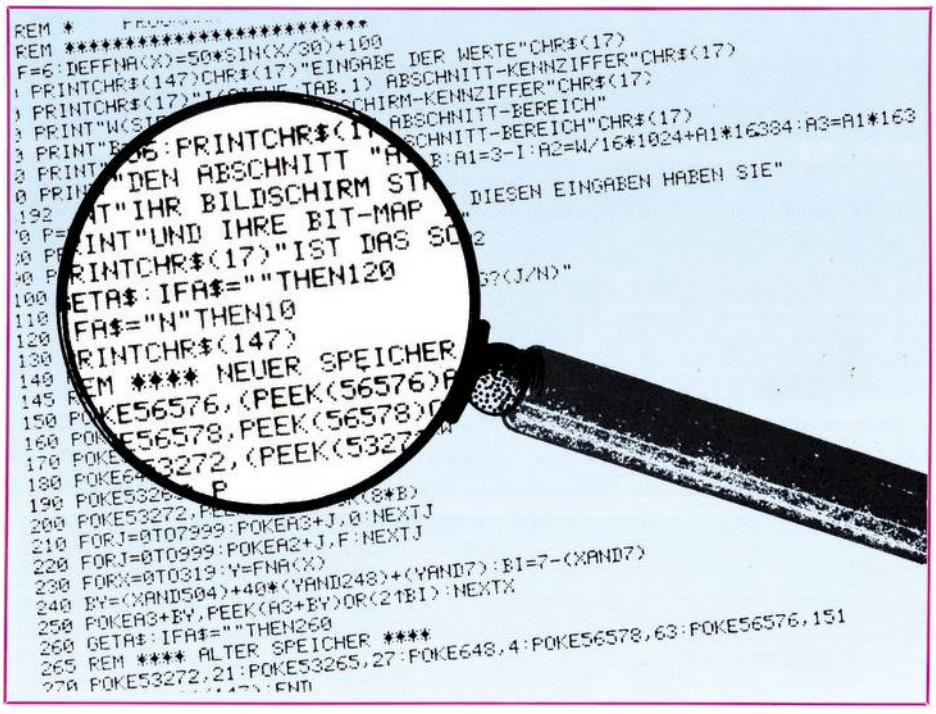
Wir erhalten wieder eine leicht zu überprüfende Liste aller DATA-Werte, diesmal aus Block 2. Bei I = 4 fällt uns sofort etwas auf. Am Bildschirm erscheint nämlich

```
I =  4   X =  15.4
```

Das ist die einzige Zahl mit Nachkommastellen, was bei dieser Art der Bildschirmausgabe sofort ins Auge sticht. Wir vergleichen den Wert mit der Eintragung im Originallisting und sehen sofort den Fehler: Wir haben beim Abtippen irrtümlich einen Punkt statt eines Kommas eingegeben. Die Korrektur ist leicht ausgeführt.

Danach löschen wir Zeile 225 mit dem GOSUB 1000 wieder und überzeugen uns durch einen abschließenden Probelauf vom einwandfreien Funktionieren des Programms.

Natürlich kann man nicht erwarten, daß sich alle Fehler so reibungslos lokalisieren lassen wie in unserem kleinen Beispiel. Gerade bei Fehlern in DATA-Zeilen kann die Suche sich namentlich bei längeren DATA-Blöcken um einiges schwieriger gestalten. Aber bei Programmen mit vielen DATA-Zeilen sind die hier beschriebenen Methoden zum Auffinden von versteckten Fehlern einfach unentbehrlich, wenn man einigermaßen schnell und sicher zum Ziel gelangen will. (ev)



```
1 REM DATA-TEST
2 REM ==========
3 REM
4 REM
5 B=1024:REM ANFANGSADRESSE BILDSCHIRM BEI C 64
6 REM
7 PRINT"^S":REM BILDSCHIRM LOESCHEN
8 PRINT:PRINT:PRINT:PRINT
9 REM
100 REM DATA-BLOCK 1 LESEN
105 REM
110 FOR I=1 TO 17
120 READ X
130 S=S+X
140 NEXT I
150 PRINT "S =";S
160 IF S<>282 THEN PRINT "PRUEFSUMMENFEHLER":END
170 PRINT "OK"
190 REM
195 REM
200 REM DATA-BLOCK 2 LESEN
205 REM
210 FOR I=0 TO 8
220 READ X
230 POKE B+I,X
240 NEXT I
250 END
290 REM
295 REM
300 REM DATA BLOCK 1
305 REM
310 DATA 12,33,11,4,17,38,22,19,7,18,
320 DATA 11,41,15,19,3,12,0
345 REM
350 REM DATA BLOCK 2
355 REM
360 DATA 3,1513,13,15.4,15,18,5

READY.
```

Listing 1: In diesem Testprogramm sind einige Fehler in den DATA-Zeilen enthalten.

```
290 REM
295 REM
300 REM DATA BLOCK 1
305 REM
310 DATA 12,33,11,4,17,38,22,19,7,18
320 DATA 11,41,15,19,3,12,0
345 REM
350 REM DATA BLOCK 2
355 REM
360 DATA 3,15,13,13,15,4,15,18,5

READY.
```

Listing 2: Dies ist nochmals der DATA-Block aus Listing 1, aber diesmal das fehlerfreie »Original«.

Listing 3: Die Bildung einer Prüfsumme ist kein zuverlässiges Mittel, um Fehler in DATA-Zeilen zu entdecken. Wie man leicht nachrechnen kann, ergibt jede der sechs DATA-Zeilen die gleiche Prüfsumme.

```
100 DATA 12,13,14,15,16,17,0
200 DATA 13,12,14,15,16,17,0
300 DATA 13,12,14,15,16,17,0,0,0
320 DATA 11,13,14,15,16,18,0
400 DATA 23,2,14,15,16,17,0
500 DATA 23,2,14,15,16,,17

READY.
```