

So macht man Programme schneller

Basic-Programme können nach drei Gesichtspunkten optimiert werden: strukturiert und lesbar, schnell sowie speicherplatzsparend. In diesem Beitrag wird der Aspekt der Laufgeschwindigkeit und ihrer Verbesserung behandelt.

Es gibt einige Dinge auf der Welt, die man sehr wohl einzeln, aber nicht alle gleichzeitig haben kann. Wirtschaftswachstum, keine Inflation und niedrige Arbeitslosigkeit lassen sich eben nicht unter einen Hut bringen.

Bei der Computerei, oder genauer gesagt beim Programmieren, gibt es in einer höheren Sprache wie Basic ebenfalls so ein magisches Dreieck: strukturierte Programme, minimaler Speicherbedarf, kürzere Laufzeiten.

Gut lesbare und klar gegliederte Programme brauchen oft viel Speicherplatz. Deswegen soll dieser Beitrag Sie zum Experimentieren anregen, mit welchen Methoden Basic-Programme schneller gemacht werden können.

Basic ist nicht immer gleich Basic

Die Commodore-Handbücher sagen leider zu diesem Thema recht wenig. Ich bitte Sie, an Ihrem C64 Platz zu nehmen.

Als erstes wollen wir uns einen einfachen aber typischen Programmablauf überlegen, welchen wir mit mehreren Basic-Möglichkeiten programmieren können. Übrigens: Wegen der guten Lesbarkeit schreiben Sie die nachfolgenden Basic-Zeilen mit Leerstellen zwischen den Befehlen.

Ich weiß, es geht auch kürzer, aber bei meinen Experimenten spielt Speicherplatz keine Rolle und außerdem habe ich noch einen Hintergedanken, den ich erst später erklären kann. Nicht zuletzt will ich dadurch auch erreichen, daß die Laufzeiten der Programme mit den Ihren übereinstimmen.

Schalten Sie bitte auch Programmierhilfen (Toolkit, Simon's Basic etc.) und Disk Operating-Programme (DOS 5.1) aus, denn sie verlangsamen den Programmablauf.

Die interne Uhr mißt die Zeit

Der C64 hat eine interne Uhr, deren Zeit abgefragt, ausgedruckt und somit zur Zeitmessung verwendet werden kann. Im Befehlssatz der Commodore-Handbücher finden Sie dazu die beiden Funktionen TI und TIS. Mein Zeitmessungsprogramm besteht aus zwei Zeilen.

Die »Stoppuhr« wird gestartet mit: 10 TIS="000000"

Sie wird am Ende des Testprogramms gestoppt mit:

```
1000 PRINT TI/60 " SEKUNDEN" :END
```

Zwischen diese beiden Zeilen stecken wir dann die Prüflinge, das heißt die Programme, deren Laufzeit wir messen wollen. Zur Prüfung geben wir zuerst die Zeile 15 mit einer simplen Verzögerungsschleife ein.

```
15 FOR T=1 TO 100:NEXT T
```

Dieses Programm hat eine Laufzeit von 0,13 Sekunden.

Das erste Testprogramm

Löschen Sie bitte wieder die Zeile 15. Als Programm, welches wir in mehreren Versionen programmieren wollen, habe ich mir einen Ablauf ausgesucht, der auch optisch verfolgbar ist. Auf dem Bildschirm soll nämlich der Buchstabe A gleich 374mal nebeneinander gedruckt werden.

Die Zahl 374 hat nichts Magisches an sich. Es sind ganz einfach 17 Zeilen voller A's auf dem Schirm. 17 Zeilen lassen uns genug Platz, um die gestoppte Zeit darunter gut lesbar anzuzeigen.

In der **Version 1** des Programms verwenden wir POKE-Befehle, mit denen wir das A und die dazugehörige Farbe auf den Bildschirm, das heißt in den Bildschirmspeicher (Video-RAM) und den Farbspeicher (Color-RAM) bringen. Beim C64 sind es die Speicherzellen 1024 und 55296. Schauen Sie bitte in den Commodore-Handbüchern nach und vergewissern Sie sich, daß Sie dieses System des Zeichen-POKEs verstehen. Es ist dort gut beschrieben. Der Buchstabe A hat den Bildschirm-Codewert 1. Als Farbe wähle ich die »Normalfarben« der beiden Computer. Der Farbcode für das Blau ist 14. Das Programm beginnt mit dem Löschen des Bildschirms (Zeile 20) und POKEt dann in Zeile 40 das erste A in den ersten Platz des Bildschirms.

```
20 PRINT CHR$(147)
```

```
40 POKE 1024+Z:POKE 55296+Z,14
```

In Zeile 40 finden Sie zusätzlich eine Variable »Z«. Wie geplant, soll das A 374 mal gePOKEt werden. Also müssen wir die Zahl der Speicherzelle laufend um 1 erhöhen. Dazu erfinden wir diese Variable Z, die zur Speicherzelle addiert wird. Wir setzen Z am Anfang des Programms (in Zeile 30) auf Null und zählen es in Zeile 50 um 1 weiter.

```
30 Z=0
```

```
50 Z=Z+1
```

Als nächstes müssen wir prüfen, ob Z den Schlußwert 374 erreicht hat. Wenn nicht, dann soll der nächste POKE-Befehl ausgeführt werden, das heißt wir springen auf Zeile 40 zurück. Dann kommt die Zeile 1000 zum Zug mit dem Stoppen und Ausdrucken der Laufzeit. Also:

```
60 IF Z=374 THEN 1000
```

```
70 GOTO 40
```

Ich schlage vor, daß Sie das kleine Programm noch mal LISTen, damit wir es komplett sehen können.

Die Laufzeit wird nur davon beeinflusst, was zwischen den Zeilen 10 und 1000 steht. Sie können vor der Zeile 10 dem Programm hinzufügen, was Sie wollen.

Ein erster Probelauf mit RUN bringt das gewünschte Ergebnis, nur eins ist noch unschön: Der PRINT-Befehl in Zeile 1000 druckt uns die Zeit oben in die 2. Zeile, wo sie schlecht erkennbar ist. Wir könnten sie in einer anderen Farbe drucken, aber ich habe einen besseren Vorschlag.

Ersatz für den Befehl »PRINT AT«

Wir brauchen ein Kochrezept, um mit einem PRINT-Befehl an einen ganz bestimmten Platz auf dem Bildschirm drucken zu können. Einige Basic-Dialekte kennen den Befehl »PRINT-AT«. Welche Möglichkeiten bietet uns das Basic von Commodore?

```
1) PRINT "{Cursor Down} {Cursor Right}"
```

```
2) PRINT TAB(X)
```

```
3) PRINT SPC(X)
```

Um zum Beispiel an den 3. Platz in der 20. Zeile die Zeit zu drucken, müßten wir 18mal das inverse Q für Cursor Down und 1mal Cursor Right eingeben.

Mit TAB (X) geht es besser. Wir haben nur ein Problem, daß nämlich der höchste zulässige Wert für X nur 255 ist (X nennt

man das »Argument«). Wir müssen deshalb zwei TAB-Befehle hintereinander setzen, um einen Abstand von 400 Leerstellen zu erzeugen.

```
1000 PRINT TAB (255) TAB (155) TI/60"SEKUNDEN":END
```

Für das Argument von SPC gilt dieselbe Begrenzung von 255. Eine doppelte Verwendung von SPC geht natürlich auch, allerdings zählt SPC nicht vom Anfang der Zeile, sondern ab der letzten Cursor-Stelle. Durch Rechnen oder einfach durch Probieren finden wir die Gesamtzahl von 397, das gibt:

```
1000 PRINT SPC (255) SPC (142) TI/60"SEKUNDEN":END
```

Es gibt noch eine dritte Methode, um PRINT-AT zu simulieren.

In die Speicherzelle 214 kann die Zahl einer Zeile hineinge-POKEt werden, auf die mit einem nachfolgenden PRINT der Cursor gesetzt wird. Das gleiche gilt für einen Platz in einer Zeile mit der Speicherzelle 211. Versuchen Sie es mit der direkten Eingabe:

```
POKE 214,8:PRINT:POKE 211,4:PRINT"A"
```

Drei Formen für »PRINT AT«

Das druckt den Buchstaben A in die 4. Spalte auf der 9. Zeile. Für unseren Anwendungsfall in Zeile 1000 müssen wir die Zahl 18 nach 214 POKEn, 211 können wir vernachlässigen.

```
1000 POKE 214,18:PRINT:PRINT TI/60"SEKUNDEN":END
```

Alle drei Methoden sind gleichwertig, sowohl in Auswirkung als auch beim Speicherbedarf. Ich bleibe im folgenden bei der 214-Methode.

Zurück zur Version 1 des Testprogramms. Das Programm unterscheidet sich für die beiden Computer nur in der Zeile 40, allerdings auch durch die Laufzeit. Nach RUN erhalten wir 10,48 Sekunden.

Dieses Auffüllen des Bildschirms mit A geht halt recht langsam. Schon beim Zuschauen wird man ungeduldig. Der Teil in diesem Programm, welcher die Laufzeit am nachhaltigsten beeinflusst, ist die 374fache Wiederholung des POKE-Befehls. Bei einem POKE-Befehl ist die Umwandlung der Zahlen aus dem ASCII-Code sehr zeitaufwendig. Hoppla, was heißt denn das schon wieder, sagen Sie jetzt vielleicht.

Jede Zahl, wie zum Beispiel 7680 oder 1024, wird zuerst als vier einzelne ASCII-Codezahlen gespeichert. Wenn das Programm abläuft, werden diese ASCII-Zahlen zuerst in ganze Zahlen, dann in Fließkomma-Zahlen umgewandelt – das für den Fall, daß mit den Zahlen arithmetische Funktionen ausgeführt werden. Schließlich werden sie wieder in eine ganzzahlige POKE-Adresse umgewandelt, und das in unserem Fall 374mal!

Hier können wir einen ersten innerbetrieblichen Verbesserungsvorschlag einreichen. Wenn wir eine so häufig vorkommende Zahl wie die POKE-Adresse in Zeile 40 am Anfang des Programms einer Variablen zuweisen, dann erfolgt die oben genannte Umwandlungssequenz nur einmal, nämlich am Anfang des Programms. Das Programm muß dann 374mal nur den Wert der Variablen im Speicher suchen, und das geht viel schneller. Wollen Sie es sehen? Ändern Sie bitte für diese **Version 2** die Zeilen 30 und 40.

Die Anfangsadresse im Bildschirmspeicher definieren wir als Variable mit dem schönen und zutreffenden Namen »VIDEO«, die des Farbspeichers mit »FARBE«. In Zeile 30 erhalten sie dann die Werte

```
30 Z=0:VIDEO=1024:FARBE=55296
```

Zeile 40 ergibt sich dann eigentlich zwangsläufig:

```
40 POKE VIDEO+Z,1:POKE FARBE+Z,14
```

Alles andere bleibt gleich. Tippen Sie RUN ein. Ergebnis: 7,3 Sekunden. Wir haben also eine Verkürzung von zirka drei Sekunden erzielt, das sind 30 Prozent!

Eine Beschleunigung von 30 Prozent ist gut, aber noch

nicht alles, was wir erreichen können. Da die Methode der vordefinierten Variablen so effektiv ist, wollen wir sie auf alle oft verwendeten Zahlen des Programms anwenden. Neben Z, VIDEO und FARBE gibt es noch die 1 für den Buchstaben A und die 14 für die Farbe sowie den Schlußwert 374 der Schleife. Sie wissen schon, wie das geht. Wir ändern folgende Zeilen:

```
30 Z=0: SCHLUSSWERT=374:VIDEO=1024:FARBE=55296:
```

```
BUCHSTA=1:DRUCK=14
```

```
40 POKE VIDEO+Z, BUCHSTA:POKE FARBE+Z,DRUCK
```

```
60 IF Z=SCHLUSSWERT THEN 1000
```

Das ist **Version 3** des Programms, mit einer Laufzeit von 6,15 Sekunden. Das ist wieder eine Verbesserung gegenüber der ersten Version.

Bisher haben wir die Variablen im Sinn einer guten Lesbarkeit mit langen und verständlichen Namen versehen. Aber das kostet natürlich Speicherplatz und auch Geschwindigkeit. Der Grund ist immer derselbe: Bei der Variablen VIDEO sind fünf Zeichen 374mal zu bearbeiten. Wenn wir sie nur VI nennen, ist das erheblich weniger. In **Version 4** des Programms reduzieren wir also alle langen Variablenamen auf zwei Zeichen. Wir wollen mal schauen, was das bringt:

```
30 Z=0: SC=374: VI=1024:FA=55296:BU=1:DR=14
```

```
40 POKE VI+Z,BU:POKE FA+Z,DR:60 IF Z=SC THEN 1000
```

Ergebnis: 5,63 Sekunden

Gegenüber der Version 1 des Programms haben wir schon eine Verbesserung von 46% erreicht.

Sie wissen sicher, daß der Computer alle Variablenamen immer auf zwei Stellen reduziert, daß aber einstellige Variablen durchaus zugelassen sind. Damit müßte unser Programm eigentlich noch schneller werden. Die folgenden Änderungen für **Version 5** zielen genau darauf:

```
30 S=374: V=1024: F=55296:B=1:D=16
```

```
40 POKE V+Z,B:POKE F+Z,D:60 IF Z=S THEN 1000
```

Die erzielte Verbesserung ist meßbar, aber nicht gerade überwältigend. Der C 64 braucht nun 5,51 Sekunden. Mit dem bisher Gesagten läßt sich bereits eine erste generelle Regel für schnellere Basic-Programme aufstellen.

Regel 1

- * Häufig vorkommende Zahlen, Adressen und Variable, besonders innerhalb von Schleifen, werden am Anfang des Programms vordefiniert.
- * Variable, die sich im Lauf des Programms verändern, werden trotzdem vordefiniert, allerdings mit einem unschädlichen Anfangswert (dummy).
- * Die Zahl, die am häufigsten vorkommt, wird als erste vordefiniert (damit sie schneller »gefunden« wird).
- * Variablenamen sollen möglichst einstellig, aber höchstens zweistellig sein.

Halt! Löschen Sie das Programm noch nicht. Wir liegen noch unter 50 Prozent mit unseren Verbesserungen, und das reicht noch lange nicht. Was können wir am bisherigen Programm noch ändern? Denken Sie mal nach. Wie kann man Buchstaben auf den Bildschirm bringen? Natürlich, mit PRINT statt der POKES.

In **Version 6** des Programms ersetzen wir die Buchstaben-POKE-Befehle durch eine PRINT-Anweisung, die prinzipiell noch den Vorteil hat, daß keine Farbanweisung nötig ist. Den Buchstaben A wollen wir in dieser Version zunächst einmal mit seiner ASCII-Codezahl 65 angeben, ein Semikolon setzt alle A hintereinander.

```
40 PRINT CHR$(65);
```

Für die Schleife brauchen wir nur die Variablen Z und S:

```
30 Z=0:S=374
```

Alle anderen Zeilen bleiben unverändert.

Nach RUN sehen wir als Ergebnis: 4,05 Sekunden

Das ergibt eine weitere Verbesserung von 1,5 Sekunden. Unser Programmbeispiel wird also durch die Verwendung von PRINT statt POKE stark beschleunigt. Das geht natürlich deswegen besonders gut, weil alle Buchstaben automatisch hintereinander gesetzt werden. Wenn wir jedesmal den Platz mit angeben müßten, wohin gePRINTet werden soll, wäre der Vorteil rasch verspielt.

Die Anweisung PRINT CHR\$(65) ist zwar gut lesbar, aber wir haben ja vorher gelernt, daß Vordefinieren von Variablen schneller ist. In **Version 7** machen wir das auch mit der PRINT-Variablen.

```
30 Z=0: S=374: A$=CHR$(65)
40 PRINT A$
```

Das Resultat ist wieder beeindruckend: 3,10 Sekunden. Das sind schon 69 Prozent Verbesserung gegenüber der 1. Version.

Aber selbst – oder gerade – jedem Anfänger ist die direkte PRINT-Anweisung mit Gänsefüßen am geläufigsten. Sie braucht auch weniger Speicherplatz und macht sogar ein Vordefinieren unnötig. Wir wollen schauen, ob sie auch schneller ist.

Ändern Sie für **Version 8** die Zeilen 30 und 40:

```
30 Z=0:S=374
40 PRINT "A";
```

Erstaunlicherweise bringt das fast gar nichts, nur 0,07 Sekunden. Die Erklärung liegt darin, daß beide Darstellungen, CHR\$(65) und "A" ASCII-Code-Verwender sind. Damit ist der einzige Unterschied zwischen Version 7 und 8 die Anzahl der Zeichen im Programm. Fassen wir zusammen:

Regel 2

- * In Schleifen mit aneinandergereihten Druckanweisungen ist PRINT viel schneller als POKE.
- * Die PRINT-Variablen sollen entweder vordefiniert oder im Gänsefuß-Modus eingesetzt werden.

In der PRINT-Schleife (Zeile 40) und nachfolgender IF-Abfrage (Zeile 60) gibt es noch zwei Feinheiten. Basic erlaubt uns bei bedingten Sprüngen statt IF .. THEN GOTO .. nur IF .. THEN .. zu schreiben, und das haben wir bisher auch brav gemacht.

Man kann aber auch IF .. GOTO verwenden. Eigentlich ist nicht zu erwarten, daß zwischen den beiden Schreibarten ein Zeitunterschied besteht. Der Fall ist tatsächlich fast akademisch, wie **Version 9** beweist:

```
60 IF Z=S GOTO 1000
```

Laufzeit des C64: 3,0 Sekunden.

Eine andere Änderung bringt auch nur ganz wenig in der Geschwindigkeit, aber sie spart uns eine ganze Zeile und damit Speicherplatz:

```
60 IF Z < > S GOTO 40
70 entfällt
```

Diese **Version 10** gewinnt nur 0,02 Sekunden.

Regel 3

- * Bei Schleifen mit Sprunganweisungen ist IF .. GOTO schneller als IF..THEN
- * Prüfung auf Ungleichheit (<>) bietet Vorteile, wenn sie eine Zeile erspart.

Für die etwas weiter Fortgeschrittenen unter Ihnen gebe ich hier noch eine weitere Anregung, die wir mit unserem Prüfprogramm nicht testen können.

Die Prüfung mit IF..THEN hängt oft von mehr als einer Bedingung ab. Zum Beispiel kann sie so lauten:

```
100 IF (A=1 AND B=2 AND C=3) THEN 999
110 GOTO 50
```

Zeile 100 prüft jedesmal, ob alle drei Bedingungen erfüllt sind, erst nach dem THEN wird entschieden, ob das Programm auf Zeile 999 springt oder auf 110 weiterläuft. Nehmen wir an, A ist im 20. Durchlauf erfüllt, B im 50. Durchlauf, C aber erst im 300. Durchlauf. Das Programm muß also 300mal alle drei Bedingungen nachprüfen. Wenn wir die Zeile 100 so schreiben:

```
100 IF C=3 THEN IF B=2 THEN IF A=1 THEN 999
```

dann bricht das Programm 300mal die Prüfung nach dem C sofort ab und geht in 110 weiter. B und A werden erst dann zur Prüfung herangezogen, wenn C=3 ist. Es ist wohl klar und einzusehen, daß die zweite Schreibweise schneller ist.

Regel 4

- Bei IF..THEN-Prüfungen mit mehreren Bedingungen sollen diese Bedingungen in einzelnen IF..THEN-Prüfungen hintereinander gesetzt werden. Dabei wird die Bedingung an die erste Stelle gesetzt, welche als erste nicht erfüllt ist.

Ich habe mit Absicht bis hierher die 374malige Wiederholung der Schleife mit der Zählvariablen Z hochgezählt und mit IF..THEN beziehungsweise IF..GOTO den Schlußwert abgefragt. Das gab mir die Gelegenheit, die Schnelligkeit dieser Methode ganz auszureizen. Und wir haben auch die ursprüngliche Laufzeit von 10,48 Sekunden auf 2,88 Sekunden reduziert! Jetzt wollen wir noch eine andere Basic-Möglichkeit austesten, die Sie ja alle kennen, nämlich die Schleife mit FOR..TO..NEXT zu programmieren.

Wir wollen in **Version 11** des immer noch gleichen Programms dazu die Zeilen 30, 50 und 60 ändern:

```
30 FOR Z=1 TO 374
50 NEXT Z
60 entfällt
```

Lassen Sie's laufen. Huiiih! Das pfeift runter! Der C64 braucht nur noch 1,08 Sekunden. Das bringt gegenüber der IF..THEN-Schleife der Version 10 eine ganze Menge, nämlich ungefähr 2 Sekunden oder, auf die Zeit der Version 1 bezogen 89 Prozent.

Das einzige, was wir in der FOR..NEXT-Schleife noch verbessern können, ist die vielgeübte Praxis des Weglassens der Schleifenvariable Z nach dem NEXT-Befehl.

```
50 NEXT
```

Der Geschwindigkeitsgewinn dieser **Version 12** ist nicht sehr groß, nämlich nur etwa 0,1 Sekunden, aber wir wollen die Methode doch in die nächste Regel mit aufnehmen.

Regel 5

- Schleifen sollen nicht mit IF..THEN, sondern mit FOR..TO..NEXT gebildet werden. Die Schleifenvariable nach NEXT soll dann weggelassen werden, wenn es nicht zu Verwechslungen mit anderen Schleifen führen kann.

Ich bin fast am Ende meines Beschleunigungslateins. Nur eines bleibt noch, nämlich die bisher hochgehaltene Lesbarkeit des Programms zu opfern. Ich hoffe nämlich, Sie haben bisher meiner Eingangsforderung Folge geleistet und alles schön mit Leerzeichen geschrieben. Das behalten wir zunächst noch bei, im Gegenteil, wir wollen zunächst die Lesbarkeit noch erhöhen und REM-Erläuterungen einfügen. Ich schlage vor, die **Version 13** so auszuschmücken:

```
10 TI$="000000":REM UHR AUF NULL
12 REM*****
13 REM*
14 REM*TEST-PROGRAMM*
15 REM*
16 REM*****
20 PRINT CHR$(147);:REM ALLES LOESCHEN
30 FOR Z=1 TO 374:REM 374 ZEICHEN
40 PRINT "A":
50 NEXT
999 REM ZEIT AUSDRUCKEN
1000 POKE 214,18:PRINT TI/60 "SEKUNDEN":END
```

Sieht gut aus, nicht wahr?

Aber leider, REM-Erläuterungen kosten Zeit. Wir sind um 0,2 Sekunden langsamer geworden.

Wir schmeißen deshalb alle REMs wieder raus und haben damit wieder Version 12. Jetzt aber gehen wir einen Schritt in der anderen Richtung weiter und entfernen alle Leerstellen und Abstände. Mit dieser **Version 14** will ich Ihnen zeigen, daß das auch einen Einfluß auf die Laufgeschwindigkeit hat.

```
10 TI$="000000"
20 PRINTCHR$(147);
30 FORZ=1TO374
40 PRINT"A";
50 NEXT
1000 POKE214,18:PRINT:PRINTTI/60"SEKUNDEN":END
```

Das Ergebnis ist 0,98 Sekunden.

In **Version 15** treiben wir die Schrumpfung ins Extrem, indem wir das Programm im Prinzip unverändert, aber mit

einem Minimum an Zeilen schreiben, also möglichst viele Befehle in eine Zeile packen.

Sie wissen, die maximale Zeilenlänge beträgt 80 Zeichen beim C64. Unser Programm können wir sogar in einer einzigen Zeile unterbringen – fast unglaublich, aber es geht. Sie müssen allerdings alle Abkürzungsmöglichkeiten ausschöpfen, die das Commodore-System bietet. Im Anhang der Commodore-Handbücher finden Sie die Liste aller Abkürzungen beim Eintippen: C und geSHIFTetes H für CHR\$, ? für PRINT und so weiter. Im nachfolgenden Ausdruck ist das natürlich nicht zu sehen, weil der Befehl LIST die Abkürzungen nicht berücksichtigt. So kommen auch mehr als 80 Zeichen in eine Zeile des Listings.

```
10 TI$="000000":PRINTCHR$(147);FORZ=1TO374:
PRINT "A";:NEXT:POKE214,18:PRINT:PRINT TI/60
"SEKUNDEN":END
```

Und siehe da, diese »Kurzform« des Programms ist auch die allerschnellste Version. Der C64 braucht 0,93 Sekunden. Diese letzte Beschleunigung wird dadurch erreicht, daß das Betriebssystem des Computers nur einmal einen Zeilenanfang und Zeilenende suchen und erkennen muß, statt sechsmal in der Version 14.

Das Ausnützen der vollen Kapazität einer Zeile bringt also nicht nur den Vorteil eines kleineren Speicherbedarfs, sondern auch Zeitgewinn.

Regel 6

* Programme ohne REM-Erläuterungen und ohne Leerstellen zwischen den Zeichen laufen schneller.

* Zur Reduzierung der Zeilenzahl sollen möglichst viele Befehle in eine Zeile geschrieben werden.

»Einzeiler« können auch Spaß und Herausforderung zugleich sein. Man sollte eigentlich annehmen, daß mit einer Zeile nicht viel anzufangen sei. Weit gefehlt!

Einzeilige Programme

Ich schreibe die Einzeiler unten lesbar, das heißt mit Leerstellen. Sie müssen die Programme aber wieder mit allen Abkürzungstricks schreiben, sonst geht's schief.

Laufzeiten der Programmversionen für »Buchstaben auf dem Bildschirm«		
Version	Programmier-Methode	Laufzeit (Sek.)
1	Buchstaben POKEn, Zählschleife mit IF ... THEN	10,48
2	POKE-Adressen vordefinieren	7,30
3	alle Variablen vordefinieren	6,15
4	Variablen mit 2 Buchstaben	5,63
5	Variable mit 1 Buchstaben	5,51
6	PRINT CHR \$ statt POKE	4,05
7	CHR \$ vordefinieren	3,10
8	PRINT "A"	3,03
9	Schleife mit IF-GOTO statt IF-THEN	3,00
10	IF Z < > S statt IF Z = S	2,98
11	Schleife mit FOR-NEXT	1,08
12	NEXT ohne Zählvariable	1,00
13	Listing mit REMs	1,20
14	Listings ohne REM, ohne Abstände	0,98
15	Alles in einer Zeile	0,93
16	Maschinensprache	0,066

Von A. Boyd (Manchester) stammt ein Primzahlen-Erzeuger, der für die obere Grenze von 65000 viele Stunden braucht.

```
1 FOR N=1 TO 65000:F=0FOR J=2 TO N-1:F=F+((N-J*INT
(N/J))=0):NEXT:X=-(F=0):PRINT RIGHT$(STR$(X*N),6*X):
NEXT
```

Ein anderer Einzeiler wurde von A.M. Simmelt (Sheffield) geschrieben zur Konvertierung von Dezimal- in Dualzahlen.

```
1 INPUT A:FOR I=14 TO 0 STEP-1:Z=A AND 2^I:I=A-Z:Z
=Z/2^I:Z$=RIGHT$(STR$(Z),1):PRINT Z$;:NEXT:
GOTO1
```

Lassen wir's gut sein mit diesem Programmsport und kehren wir zurück zu einer abschließenden Betrachtung der Zeitgewinne.

Wir haben in Version 1 mit 10,48 Sekunden begonnen. Diese Laufzeit wurde ohne Änderung des Programmresultats stetig verkürzt, bis wir schließlich in Version 15 bei 0,93 Sekunden gelandet sind. Ich nenne diese Beschleunigung um 90 Prozent schlicht und einfach spektakulär.

Mehr allerdings kann ich nicht herausholen, es sei denn – na ja, eigentlich habe ich am Anfang ganz laut »Basic« gesagt. Aber ich kann doch nicht widerstehen und Sie scharf machen auf ultima velocitas – zu deutsch Maschinensprache.

Es soll Ihnen aber kein Maschinensprache-Kurs zugemutet werden. Doch ein Programm in Maschinensprache besteht genauso aus Befehlen, Adressen und Variablen wie ein Basic-Programm, nur sind sie in einem speziellen Zahlencode geschrieben. Dieser Zahlencode muß in den Arbeitsspeicher geladen werden. Die für uns einfachste Möglichkeit besteht darin, die Zahlen in den Speicher hineinzupOKEn. Damit wir aber nicht unmäßig viele POKE-Befehle schreiben müssen, legen wir alle Code-Zahlen hinter DATA-Befehle und holen sie dann mit READ in eine einzige POKE-Schleife. Ich sage das deswegen, weil dieses Einlesen natürlich nicht zu dem Testprogramm gehören darf, dessen Laufzeit wir messen wollen. Das Testprogramm selbst sitzt zwischen den drei Zeilen der »Stoppuhr«. Das heißt, genauer gesagt sitzt das Programm in den Speicherzellen, in die wir es hineinPOKEN. Aber zwischen der Stoppuhr rufen wir es auf, der dem RUN entsprechende Befehl bei Maschinensprache heißt SYS.

Wie Sie gleich noch sehen werden, fängt unser Testprogramm ab Speicherzelle 7168 an. Das Ganze sieht dann so aus:

```
10 TI$="000000"
20 PRINT CHR$(147)
30 SYS 7168
1000 POKE 214,18:PRINT:PRINT TI/60 "SEKUNDEN":END
```

Ab Zeile 2000 setzen wir jetzt das Programm, welches uns das Maschinenprogramm einliest. Um mit dem Einlesen zu beginnen, setzen wir noch eine Umleitung vor das Meßprogramm: 5 GOTO 2000

In Zeile 2000 löschen wir den Bildschirm. Zeile 2010 und 2020 und 2030 liest die Codezahlen, die von Zeile 2050 bis 2090 stehen, und POKet sie in die Speicherplätze 7168 bis 7200. Sobald die Zahlen eingelesen sind, können Sie das Meßprogramm mit dem Befehl GOTO 10 (direkt eingetippt) starten.

Im Abdruck unten wird das etwas eleganter gemacht. Zuerst meldet das Programm das Ende des Einlesens (Zeile 2100 und 2101). Dann kommt die Anweisung, wie das Meßprogramm zu starten ist, nämlich durch Drücken irgendeiner Taste, die durch eine GET-Schleife abgefragt wird. Wenn eine Taste gedrückt wird, springt das Programm auf Zeile 10 (das geschieht in Zeile 2160).

```

5 GOTO 2000
10 TI$="000000"
20 PRINT CHR$(147)
30 SYS 7168
1000 POKE 214,18:PRINT:PRINT TI/60 "SEKUNDEN":END
2000 PRINT CHR$(147)
2010 FOR A=7168 TO 7200
2020 READ B
2030 POKE A,B
2040 NEXT
2050 DATA 162,0,169,1,157
2060 DATA 0,4,169,14,157,0,216
2070 DATA 232,224,0,208,241,169,1,157
2080 DATA 254,4,169,14,157,254,216
2090 DATA 232,224,120,208,241,96
2100 PRINT "DAS MASCHINENPROGRAMM"
2110 PRINT "IST JETZT EINGELESEN.":PRINT
2120 PRINT "ZUM STARTENDES PRO-"
2130 PRINT "GRAMMS" CHR$(18) "TASTE" CHR$(146)
"DRUECKEN"
2140 GET A$
2150 IF A$=" " THEN 2140
2160 GOTO 10

```

So, inzwischen haben Sie sicher Ihre Überraschung gehabt! 0,066 Sekunden Laufzeit. Ich hoffe, daß ich Sie mit dem Virus der Maschinensprache infiziert habe.

Wir wollen im folgenden ein paar arithmetische Funktionen untersuchen und beschleunigen. Als erste nehmen wir uns in **Version 17** die Multiplikation vor. Die Messung der Laufzeit erfolgt auf dieselbe Weise wie bei allen Programmen vorher auch. Deshalb bleiben die Zeilen 10, 20 und 1000 gleich. Die Multiplikation selbst soll 300mal ausgeführt werden (Zeile 30). Dann wird das Ergebnis gedruckt (Zeile 60).

```

30 FOR Z=1 TO 300
50 NEXT
60 PRINT A

```

Als Multiplikation nehmen wir den Extremfall einer kurzen Zahl multipliziert mit einer langen.

```
40 A=3*0,123456789
```

Nach RUN bleibt der Bildschirm zuerst leer, bis dann nach 11,85 (14, 15) Sekunden das Ergebnis der Multiplikation und die Laufzeit ausgedruckt wird.

In **Version 18** vertauschen wir die beiden Zahlen, die in Zeile 40 multipliziert werden.

```
40 A=0.123456789*3
```

Diese einfache Manipulation bringt natürlich nach Adam Riese dasselbe Resultat wie vorher, aber die Laufzeit ist kürzer. Wir gewinnen 0,44 Sekunden. Dieser Gewinn ist nicht überwältigend, aber überraschend. Aber denken Sie nach!

Wie ist das, wenn Sie so eine Multiplikation auf dem Papier durchführen? Da ist die Rechnung im zweiten Fall auch einfacher. Der Computer hat genau dasselbe Problem.

In **Version 19** nützen wir noch eine kleine Eigenheit der Commodore-Computer aus, die auf ihre amerikanische Herkunft zurückzuführen ist. Bei den Angelsachsen ist es nämlich erlaubt, eine Null vor dem Dezimalpunkt wegzulassen. Beim Computer dürfen wir das auch. Obwohl das mit der Multiplikation direkt nichts zu tun hat, bietet sie uns doch eine gute Gelegenheit, die Einsparung durch das Weglassen der Null auch zeitlich zu messen. Also, Zeile 40 sieht jetzt so aus:

```
40 A=.123456789*3
```

Das bringt nicht sehr viel, 0,20 Sekunden. Aber Kleinvieh macht auch Mist.

Eine ähnliche Verbesserung, die wir hier nicht ausprobieren, wird erzielt durch den Ersatz einer alleinstehenden Null durch einen Punkt, zum Beispiel:

statt IF X=0 THEN -
jetzt IF x=. THEN -

Eine gewaltige Beschleunigung erfährt das Multiplikationsbeispiel, wenn wir die Regel 1 anwenden und die Variablen vordefinieren.

Regel 7

* Bei einer Multiplikation soll die längere Zahl vor der kürzeren stehen (langer Multiplikant, kurzer Multiplikator).

* Eine einzelne Null wird durch einen Punkt ersetzt, eine Null vor dem Dezimalpunkt wird weggelassen.

In **Version 20** ersetzen wir in Zeile 40 beide Zahlen durch Buchstaben, die wir in einer neuen Zeile 25 diese Werte zuweisen.

```
25 B=.123456789:C=3
40 A=B*C
```

Dieser Lauf bleibt nach 1,48 Sekunden stehen, das heißt wir gewinnen 12,23 Sekunden. Also bitte Regel 1 unbedingt beachten!

Eine andere betrachtenwerte arithmetische Funktion ist das »Potenzieren« (Quadrat-/Kubikzahlen), ausgelöst durch das Zeichen **↑**. **Version 21** erzielen wir durch Löschen der Zeile 25 und Abänderung der Zeile 40:

```
40 A= 4 ↑ 3
```

»Vier hoch drei« ergibt 64 und braucht 10,53 Sekunden.

In **Version 22** wollen wir sehen, ob vordefinierte Variable auch so einschlagen wie bei der Multiplikation.

```
25 B=4:C=3
40 A=B↑C
```

Man kann sich doch auf nichts verlassen! Diesmal sind wir nur um 0,22 Sekunden schneller. Wir dürfen aber nicht aufgeben. **Version 23** macht alles wieder wett, und zwar durch den simplen Trick, daß wir das Potenzieren in seine Grundelemente zerlegen.

Sie wissen doch: 4 hoch 3 (4↑3) ist dasselbe wie »4 zweimal mit sich selbst multipliziert« (4*4*4).

```
25 B=4 (C entfällt)
40 A=B*B*B
```

Ja, da schauen Sie, geht? Das Programm braucht nur 1,68, also 8,31 Sekunden weniger.

Regel 8

Die Funktion Potenzieren (↑) soll durch Mehrfach-Multiplikation ersetzt werden.

Als letztes Objekt möchte ich oft aufgerufene Unterprogramme messen. Wir erreichen das ganz einfach dadurch, daß wir das letzte Programm (Version 23) abändern. So erhalten wir **Version 24**: Die Definition der Variablen (Zeile 25) und die Multiplikation (Zeile 40) verbannen wir als Unterprogramm an das Ende des Programms und springen innerhalb der 300fachen Schleife mit GOTO darauf.

```
25 löschen;
30 FOR Z=1 TO 300;
40 GOTO 40000
```

Alles andere bleibt, aber neu kommt dazu: 40000 B=4;

```
50000 A=B*B*B;
60000 GOTO 50
```

Es ist nicht weiter erstaunlich, daß dieser Umbau diese Version 24 gegenüber Version 23 verlangsamt. Aber merken Sie sich die Laufzeit, 3,28 Sekunden. Als nächstes ersetzen wir die beiden GOTO-Zeilen durch GOSUB-RETURN.

```
40 GOSUB 40000
60000 RETURN
```

Diese **Version 25** spart uns 0,15 Sekunden. GOSUB ist schneller als GOTO! Sie haben vielleicht schon gelesen, daß oft gebrauchte Unterprogramme am Anfang eines Programms stehen sollen. Den Grund dafür will ich Ihnen mit den nächsten zwei Versionen vorführen.

Version 26 macht das zunächst für die GOTO-Version. Wir bauen sie auf der Version 24 auf, mit folgenden Änderungen:

Die Zeitmessung lassen wir wie gehabt in den Zeilen 10, 20 und 1000, die Schleife und den Ausdruck des Resultats in den Zeilen 30, 50 und 60.

Nur beim Unterprogramm streichen wir alle Nullen der Zeilennummern, so daß es jetzt in den Zeilen 4, 5 und 6 steht. Um zu vermeiden, daß das Programm gleich mit dem Unterprogramm beginnt, fügen wir davor (Zeile 3) noch eine Umleitung ein, die sofort auf der Zeile 10 weitermacht. Schließlich brauchen wir noch den Sprung in das Unterprogramm, den wir in die Zeile 33 setzten. Das Ganze sieht jetzt so aus:

```
3 GOTO 10
4 B=4
5 A=B*B*B
6 GOTO 50
10 TI$="000000"
20 PRINT CHR$(147)
30 FOR Z=1 TO 300
33 GOTO 4
50 NEXT
60 PRINT A
1000 POKE 214,18:PRINT:PRINT TI/60 "SEKUNDEN":END
```

Nach RUN erhalten wir 3,1 Sekunden. Gegenüber Version 24, unserem Vergleichsobjekt, sparen wir 0,18 Sekunden.

Dasselbe passiert, wenn wir in der **Version 27** die GOTOS mit GOSUB-RETURN ersetzen.

```
6 RETURN
33 GOSUB 4
```

Gegenüber der anderen GOSUB-Version (Version 25) sparen wir 0,17 Sekunden.

Regel 9

* Der Aufruf von Unterprogrammen mit GOSUB ist schneller als mit GOTO.

* Häufig gebrauchte Unterprogramme gehören ganz an den Anfang eines Programms. Sie müssen dann allerdings zuerst mit einem GOTO umgangen werden.

Ich bin überzeugt, daß in Basic noch mehr spektakuläre Zeitgewinne stecken. Falls Sie eine Regel 10 oder noch mehr entdecken, ermuntere ich Sie um Mitteilung.

Wenn Sie Fragen haben, können Sie mich mit einer Leserzuschrift ansprechen. (Dr. Helmuth Hauck/aa/hm)

Version	Programmier-Methode	Laufzeit (Sek.)
17	Multiplikation, lang x kurz	14,15
18	Multiplikation, kurz x lang	13,71
19	Null weglassen	13,51
20	Variable vordefinieren	1,48
21	Potenzieren (4 hoch 3) mit 1	10,53
22	Variable vordefinieren	10,31
23	4 x 4 x 4 statt 413	2,01
Laufzeiten der Versionen für arithmetische Funktionen		

24	Unterprogramm am Ende, Sprung mit GOTO-GOTO	3,28
25	Unterprogramm am Ende, Sprung mit GOSUB-RETURN	3,13
26	Unterprogramm am Anfang, Sprung mit GOTO-GOTO	3,10
27	Unterprogramm am Anfang, Sprung mit GOSUB-RETURN	2,96
Laufzeiten der Versionen für Unterprogramme		

Debugging - Fehlersuche in Basic- Programmen

Diese Situation kennt wohl jeder Besitzer eines Heimcomputers 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 nicht so, wie es eigentlich sollte.

Das Spektrum der möglichen Ereignisse reicht in einem solchen Fall 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 Retten des mühevoll erstellten Programms 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.

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 allgemeingül-