Effektives Programmieren in Assembler

Es gibt viele Möglichkeiten, ein Basic-Programm schneller und komfortabler zu gestalten. Aber auch für die Assemblerprogrammierung gibt es einige Tricks und Kniffe, die wir Ihnen in diesem praxisnahen Kurs verraten wollen.

Wer das Optimum an Geschwindigkeit aus seinem Computer herausholen will, kommt an Maschinensprache nicht vorbei. Die Grundlagen zur Maschinenprogrammierung wurden bereits im Kurs »Assembler ist keine Alchimie«, den Sie in diesem Sonderheft finden, geschaffen. Das Thema dieses Artikels ist es nun, die Möglichkeiten von Maschinensprache optimal zu nutzen. Sie erfahren, wie man

a) Programme beschleunigen und
b) Speicherplatz sparen kann.

Dazu werden Ihnen eine Vielzahl von Programmiertechniken, Tips und Tricks vermittelt, die Ihnen die Programmierung erleichern.

1. Beschleunigungen des Betriebssystems (in Assembler)

Der C 64 muß viele Aufgaben gleichzeitig erledigen: Bearbeiten des Hauptprogramms, Ablauf der Systeminterrupts und Senden des Video-Signals (an den Monitor/Fernseher). Alle diese Funktionen erfordern

- viele Zugriffe auf den Datenbus des Prozessors
- und dadurch Ausführungszeit.

Unser Grundproblem ist nun, wie wir den Computer dazu bewegen, diese Aufgaben nicht (oder nur teilweise) auszuführen.

a) Eingriffe in den Systeminterrupt

Eine detaillierte Beschreibung des Systeminterrupts finden Sie im bereits erwähnten Kurs »Assembler ist keine Alchimie«. Hier möchte ich nur zusammenfassen, was im normalen Interrupt des Betriebssystems geschieht: 60 mal in der Sekunde wird das Hauptprogramm verlassen und die Routine ab $EA31 angesprungen. Ist diese abgearbeitet, wird wieder ins Hauptprogramm zurückgesprungen. Während dieser Unterbrechung (»interrupt«) tut sich einiges:

- die RUN/STOP-Taste wird überprüft
- die Tastatur und der Datasettenmotor werden abgefragt
- das Cursorblinken wird erledigt
- die interne Uhr (TI$) wird gestellt.

Überlegen wir uns, welche Funktionen verzichtbar sind: Die RUN/STOP-Taste bewirkt nur in Basic-Programmen einen Abbruch, in Assembler müßte sie zum Beispiel über »JSR $FFE1« zusätzlich abgefragt werden. Die interne Uhr findet von Maschinensprache aus praktisch keine Verwendung. Kurz und gut, ein Maschinenprogramm kann auf beide Funktionen verzichten. Dies wird durch ein

```
LDA  #$34
STA  $0314
```

erreicht. Weil der Computer dadurch entlastet wird, läuft das Hauptprogramm etwas schneller ab.

Die Normaleinstellung erhält man mit

```
LDA  #$31
STA  $0314
```

Beschleunigungsmethode 1:
Trick: Verkürzung der Interrupt-Routine
Nebenwirkungen: Abfrage der STOP-Taste und interne Uhr entfallen

Können Sie zwischenzeitlich auf die ganze Interrupt-Routine verzichten, genügt ein einziger Befehl:

SEI (»set interrupt«)

Er verhindert grundsätzlich das Auftreten von Interrupts. Die Normaleinstellung bewirkt:

CLI (»clear interrupt«)

Beschleunigungsmethode 2:
Trick: Interrupt total abschalten
Nebenwirkungen: Abfrage von Tastatur, STOP-Taste und Datasette, sowie Cursor und interne Uhr entfallen.

Es gibt aber noch eine Möglichkeit, im Zusammenhang mit dem Systeminterrupt: Von der Adresse $DC05, die als Zähler dient, hängt die Anzahl der Interrupts (in der Regel 60 Aufrufe pro Sekunde) in einer bestimmten Zeit ab. Diese Adresse kann durch Schreibzugriff geändert werden. Schreibt man in $DC05 einen niedrigen Wert (im Extremfall 0), so werden sehr viele Interrupts ausgelöst. Dies macht sich in der Geschwindigkeit der Interrupt-Routine bemerkbar. Cursor und Tastaturabfrage werden sehr schnell, die interne Uhr geht vor, und so weiter. Verwendet man eine eigene, eventuell zeitkritische Interrupt-Routine, kann sie auf diese Weise beschleunigt werden.

Dieser Geschwindigkeitszuwachs geht allerdings auf Kosten des Hauptprogramms, das stark verlangsamt wird. Bei wenigen Interrupts (große Zahl in $DC05) wird es beschleunigt. Die entsprechenden Assemblerbefehle lauten:

```
LDA  #$FF
STA  $DC05
```

um eine starke Beschleunigung zu bewirken.
Die Normaleinstellung wird durch

```
LDA  #$3A
STA  $DC05
```

erreicht.

Beschleunigungsmethode 3:
Trick: Anzahl der Interruptaufrufe pro Sekunde ändern
Nebenwirkungen: Bei zu wenigen Aufrufen hinken Uhr, Cursor und Tastaturabfrage nach; bei zu vielen werden sie zu schnell.

b) VIC-Register Nummer 17

Ist Ihnen schon bei Hypra-Load, beim Arbeiten mit der Datasette und einigen Kopierprogrammen aufgefallen, daß manchmal der Bildschirm abgeschaltet wird (ähnlich wie im FAST-Mode des C 128)? Dies kann man mit einem Vorhang vergleichen, der zwischenzeitlich den Bildschirm verdeckt. Der Bildschirm kann zwar nach wie vor (hinter dem Vorhang) geändert werden (PRINT-Anweisungen werden also ausgeführt), aber sichtbar wird die Wirkung erst, wenn der Vorhang entfernt wird.

Verantwortlich für das Ein-/Ausschalten des Bildschirms ist das VIC-Register Nummer 17:

Bit 4 gesetzt: Bildschirm wird angezeigt
Bit 4 gelöscht: Bildschirm wird abgeschaltet und nimmt Rahmenfarbe an.

Da wir die theoretischen Grundlagen haben, brauchen wir nur noch unser Wissen in Befehle umzusetzen:
Bildschirm abschalten:

```
LDA  $D011   ($D011 ist VIC-Register #17)
AND  #$EF    ($EF = %11101111)
STA  $D011                 ↑
                          Bit 4
```

In diesem Zustand arbeiten manche Kopierprogramme um zirka 15% schneller. Programme, die nicht auf externe Geräte wie die Floppy zugreifen, laufen zirka 5% schneller ab.
Bildschirm wieder einschalten:

```
LDA  $D011
ORA  #$10    ($10 = %00010000)
STA  $D011                 ↑
                          Bit 4
```

Dies ist der Normalzustand.

Beschleunigungsmethode 4:
Trick: Bildschirm abschalten
Nebenwirkungen: Der Bildschirminhalt ist nicht zu sehen, geht aber auch nicht verloren.

c) Hinweise zum bisher Gesagten

Alle bis zu dieser Stelle genannten Tricks beziehen sich auf die Beschleunigung von Programmen. Sie lassen sich leicht nachträglich einfügen, weil am Programmalgorithmus keine Änderungen erforderlich sind.

Sie können das Abschalten des Bildschirms mit dem Abschalten oder Einschränken des Interrupts verknüpfen, um die Geschwindigkeit noch weiter zu erhöhen. Wenn Sie den Interrupt ganz abschalten (SEI), bringt es keinen zusätzlichen Gewinn, ihn einzuschränken oder die Zahl der Aufrufe zu ändern.

Beachten Sie bitte, daß alle beschriebenen Tricks durch RUN/STOP-RESTORE, einem Reset oder den Assemblerbefehl BRK rückgängig gemacht werden.

2. Systembeschleunigungen in Basic

Hier erfahren Sie, wie sich die Systembeschleunigungen von Basic aus verwerten lassen. Die Nebenwirkungen bleiben allerdings diegleichen, wie unter 1. genannt.

a) Interrupt einschränken

POKE 788,52 verkürzt die Interrupt-Routine um das Abfragen der RUN/STOP-Taste und das Stellen von TI$.

POKE 788,49 Normalzustand

In Basic ist das Ausfallen von RUN/STOP und TI$ wesentlich störender als in Maschinensprache. Überprüfen Sie daher Ihre Programme auf Verwendung von TI$ und fügen Sie den POKE erst nach (!) der Fertigstellung des Programms ein.

b) Interrupt abschalten

POKE 56334,PEEK (56334) AND 254

schaltet den Interrupt ab,

POKE 56334,PEEK (56334) OR 1

schaltet ihn wieder ein. Dies geschieht dadurch, daß der Timer ab- beziehungsweise wieder eingeschaltet wird.

c) Anzahl der Interrupt-Aufrufe ändern

POKE 56325,0: Extrem viele Interruptaufrufe
POKE 56325,255: Extrem wenige (daraus folgt: Interrupt langsam, Basic-Programm schnell)

d) Bildschirm abschalten

POKE 53265,PEEK(53265) AND 239

schaltet den Bildschirm ab.

POKE 53265,PEEK(53265) OR 16

schaltet ihn wieder ein.

An dieser Stelle sei noch einmal auf Punkt 1c hingewiesen, damit keine (vermeidbaren) Probleme auftreten.

Anhand von Listing 1 wollen wir uns nun mit der Anwendung der Systembeschleunigungen befassen. Dieses kleine Beispielprogramm, an dem Sie nach Herzenslust experimentieren können, versucht, mit Hilfe von TI$ die Arbeitsdauer der Schleife (Zeile 150) zu messen.

Während des Ablaufs dieser Schleife, die kontinuierlich die Rahmenfarbe ändert, sollten Sie keine Taste drücken, um die Meßwerte nicht zu verfälschen.
Wenn Sie dies beachten, erhalten Sie folgende Werte:

1. Normalzustand: 000003
2. Verkürzter Interrupt: 000000
   An der gemessenen Zeit können Sie erkennen, daß TI$ abgeschaltet wurde.
3. Häufige Interrupts: 000010
   Aufgrund vieler Interrupt-Anforderungen wurde die Uhr TI$ sehr oft erhöht.
4. Seltene Interrupts: 000001
   Da die IRQ-Routine nur selten durchlaufen wurde, ist TI$ kaum weitergezählt worden.
5. Bildschirm abgeschaltet: 000002

Nur bei diesem Punkt (und natürlich auch bei »1«) hat TI$ volle Aussagekraft bezüglich der Ablaufzeit. An dieser Zeit können wir erkennen, daß durch das Abschalten des Bildschirms tatsächlich gegenüber »1« ein Zeitgewinn anfällt.

Bei den Punkten »3« und »4« wurde der Cursor eingeschaltet. Bei »3« (häufige Interrupts) ist er sehr schnell, bei »4« dagegen sehr langsam.

An Punkt »5« können Sie erkennen, daß bei abgeschaltetem Bildschirm der Hintergrund immer die Rahmenfarbe ($D020) annimmt, ohne daß wir die entsprechende Farbe ins Register $D021 »POKEn«.

```
90 GOTO 200                                          <026>
100 REM >> UP - SCHLEIFE <<                          <138>
110 :                                                <086>
120 PRINT" <TASTE>";:WAIT 198,1:POKE 198,0
    :FOR I=1 TO 7:PRINT CHR$(20);:NEXT               <221>
130 :                                                <106>
140 FOR I=1 TO 100:NEXT                              <122>
150 TI$="000000":FOR I = 0 TO 255:POKE 532
    80,I AND 15:NEXT:PRINT TI$:RETURN                <205>
160 :                                                <136>
170 REM >> UP - CURSORBLINKEN AUS <<                 <206>
180 :                                                <156>
190 POKE 207,0:POKE 204,1:PRINT" ":RETURN            <186>
200 REM --------------------                         <222>
210 REM -- HAUPTPROGRAMM --                          <045>
220 REM --------------------                         <242>
230 :                                                <206>
240 PRINT CHR$(147)"DEMO FUER SYSTEMBESCHL
    EUNIGUNGEN (BASIC)";                             <061>
250 PRINT"----------------------------------
    --------"                                        <127>
260 PRINT"{2DOWN}1) NORMALZUSTAND";:GOSUB
    100                                              <033>
270 :                                                <248>
280 PRINT"{DOWN}2) VERKUERZTER INTERRUPT";
    :POKE 788,52:GOSUB 100:POKE 788,49               <084>
290 :                                                <012>
300 PRINT"{DOWN}3) HAEUFIGE INTERRUPTS";:P
    OKE 56325,20:POKE 204,0:GOSUB 100:GOSU
    B 170                                            <018>
310 :                                                <032>
320 PRINT"4) SELTENE{2SPACE}INTERRUPTS";:P
    OKE 56325,150:POKE 204,0:GOSUB 100:GOS
    UB 170                                           <113>
330 SYS 64931:REM NORMALZUSTAND EIN                  <253>
340 :                                                <062>
350 PRINT"5) BILDSCHIRM ABGESCHALTET ";:PO
    KE 53265,PEEK(53265) AND 239:GOSUB 140           <107>
360 POKE 53265,PEEK(53265) OR 16:PRINT"{DO
    WN}** ENDE **"                                   <066>

© 64'er
```

Listing 1. Systembeschleunigungen in Basic

3. Optimierung der Bildschirmausgabe

Ohne die Bildschirmausgabe kommt kein Programm aus, aber oft kostet sie unnötig viel Rechenzeit. Der Grund ist hier nicht beim Betriebssystem zu suchen, sondern bei umständlicher Programmierung. Diese wiederum ist auf mangelndes Know-how zurückzuführen, welches wir nun ändern wollen.

In der Regel wird zur Ausgabe eines Zeichens dieses in den Akku geladen und die Routine BASOUT ($FFD2) aufgerufen. Veranschaulichen wir uns einmal die Arbeitsweise von BASOUT: Das Betriebssystem prüft bei jedem Zeichen, ob es sich um einen Buchstaben oder ein Steuerzeichen, zum Beispiel »Bildschirm löschen« handelt. Buchstaben werden in den Bildschirmcode umgewandelt und ins Bildschirm-RAM ab $0400 geschrieben.

Für Steuerzeichen existieren jeweils Unterroutinen die zum Beispiel eine Leerzeile einfügen, den Bildschirm löschen oder ähnliches.

Diese aufwendige Überprüfung verlangsamt die Bildschirmausgabe erheblich. BASOUT läßt sich zwar geringfügig beschleunigen, indem man statt bei $FFD2 (Kerneleinsprung) bei $E716 einsteigt, aber es geht noch schneller:

a) Bildschirm löschen

Langsam:

```
LDA #$93   $93 = 147 = Code für »Bildschirm
           löschen«, entspricht PRINT CHR$(147)
JSR $FFD2  (oder $E176)
```

Schnell:

```
JSR $E544  (Routine für »Bildschirm löschen«)
```

b) Cursor in Home-Position (linke obere Ecke)

Langsam:

```
LDA #$13 ; $13 = Code für »Cursor Home«
JSR $FFD2  (oder $E176)
```

Schnell:

```
JSR $E566  (Routine für »Cursor Home«)
```

c) Cursor-Positionierung

Langsam:

Senden von Steuerzeichen (CRSR DOWN, UP und so weiter) über BASOUT.

Schnell:

```
LDX #Zeile
LDY #Spalte
JSR $E50C (Cursorposition setzen)
```

d) Textausgabe

Unkomfortable Lösung:

Senden von Zeichen (Buchstaben, Grafikzeichen) über BASOUT.

Eine solche Schleife finden Sie in Listing 2, Zeilen 148 - 220 und 320 - 330. Nach dem Start durch »SYS 49152« gibt Listing 2 zweimal hintereinander den Text »DAS IST DER TEXT« aus. Das erste Mal wird der Text über eine BASOUT-Schleife gedrückt, beim zweiten Mal nimmt das Programm die Komfortable Lösung:

Ab der Adresse »TEXT« muß der Text (in ASCII-Darstellung) stehen, in dem keine Anführungszeichen vorkommen dürfen. Am Ende des Textes muß $00 als Endmarkierung zu finden sein. Die Ausgabe erfolgt dann über

```
LDA # <(TEXT)     Low-Byte der Adresse
LDY # >(TEXT)     High-Byte
JSR $AB1E
```

Die Routine $AB1E wird fortan als »STROUT« (STRing-OUTput = String-Ausgabe) bezeichnet. STROUT ist zwar etwas langsamer als BASOUT; dafür erlaubt die komfortable Parameterübergabe eine wesentlich bequemere Programmierung, wie Sie am zweiten Teil von Listing 2 (Zeilen 260 - 300, 320 - 330) sehen können. Mit nur drei Befehlen wird der Text ausgegeben!

Beschleunigungsmethode 5.

Zusammenfassung der bisherigen Alternativen zu BASOUT:

CLEAR HOME:	JSR $E544
CURSOR HOME:	JSR $E566
Cursorpositionierung:	LDX #Zeile
	LDY #Spalte
	JSR $E50C
Textausgabe:	Text ab TEXT ablegen
	(wie Listing 2, Zeile 320 - 330)
	LDA # <(TEXT)
	LDY # >(TEXT)
	JSR $AB1E

Alle diese Verfahren sind nicht nur schnell, sondern auch speicherplatzsparend.

Eine Anwendung von (fast) allen Routinen aus der Beschleunigungsmethode 5 zeigt Listing 3.

```
100  -.LI 1,3,0
110  -;
120  -; TEXTAUSGABE (UEBER BASOUT)
130  -;
140  -.BA $C000 ; START: SYS 49152
150  -;
160  -.GL BASOUT = $FFD2
170  -;
180  -          LDX #0
190  -SCHLEIFE  LDA TEXT,X      ; ZEICHEN LESEN
200  -          INX
210  -          JSR BASOUT      ; UND AUSGEBEN
220  -          BNE SCHLEIFE    ; SCHON ENDMARKIERUNG?
230  -;
240  -; TEXTAUSGABE (UEBER STROUT)
250  -;
260  -.GL STROUT = $AB1E
270  -;
280  -          LDA #<(TEXT)    ; LOW-BYTE IN AKKU
290  -          LDY #>(TEXT)    ; HIGH-BYTE IN Y
300  -          JMP STROUT      ; TEXTAUSGABE UND ENDE
310  -;
320  -TEXT      .TX "DAS IST DER TEXT!"
330  -.BY 0 ; ENDMARKIERUNG DES TEXTES
```

Listing 2. Die unkomfortable Lösung, einen Text auszugeben

```
SEARCHING FOR $$

100  -.LI 1,3,0
110  -;
120  -; TEXTAUSGABE (UEBER STROUT)
130  -;
140  -.BA $C000 ; START: SYS 49152
150  -;
160  -.GL STROUT = $AB1E
170  -.GL CURSOR = $E50C
180  -.GL CLRSCR = $E544 ; BILDSCHIRM LOESCHEN
190  -;
200  -.GL ZEILE  = 12
210  -.GL SPALTE = 10
220  -;
230  -          JSR CLRSCR      ; = PRINT CHR$(147)
240  -          LDX #ZEILE      ; ZEILE IN X
250  -          LDY #SPALTE     ; SPALTE IN Y
260  -          JSR CURSOR      ; CURSOR SETZEN
270  -          LDA #<(TEXT)    ; LOW-BYTE IN AKKU
280  -          LDY #>(TEXT)    ; HIGH-BYTE IN Y
290  -          JMP STROUT      ; TEXTAUSGABE & ENDE
300  -;
310  -TEXT      .TX "DAS IST DER TEXT!"
320  -.BY 0 ; ENDMARKIERUNG FUER STROUT
```

Listing 3. Die komfortable Lösung einen Text auszugeben

Der Bildschirm wird gelöscht und in Zeile 12 ab Spalte 10 ein Text ausgegeben. Auch in diesem Programm sollten Sie zur Übung etwas experimentieren!

e) Kopieren des Textes in den Bildschirmspeicher

Dies ist die schnellste Methode: Der Text wird in den Bildschirmspeicher kopiert. Die lange Umwandlung entfällt völlig, da der Text als fertiger Bildschirmcode im Speicher abgelegt wird. Wenn einige Kopfzeilen (zum Beispiel mit Copyright-Vermerken) an verschiedenen Stellen ausgegeben werden sollen, ist es ratsam, ein kleines Unterprogramm zu erstellen. Dieses schreibt dann die Kopfzeilen direkt in den Bildschirmspeicher, ohne die aktuelle Cursor-Position zu beeinflussen.

Eines müssen Sie aber unbedingt beachten: Die Farbgebung ist nur durch Ändern des Farb-RAMs möglich.

Eine Tabelle der Bildschirmcodes finden Sie übrigens im Anhang des C 64-Handbuchs und am Schluß dieser Ausgabe.

Beschäftigen wir uns nun mit Listing 4:

Dieses Programm entspricht in der Wirkung Listing 3, gibt den Text jedoch nicht über die Betriebssystem-Routinen CURSOR und STROUT aus, sondern schreibt ihn direkt in den Bildschirm.

In den Zeilen 310 - 320 steht der Bildschirmcode des Textes.

Zurück zur Routine STROUT: Diese Routine arbeitet, da sie sich auf die BASOUT-Routine stützt, auch mit Peripheriegeräten wie Floppy und Drucker, wenn diese über dem CMD-Befehl als Ausgabegeräte definiert wurden. In »Assembler ist keine Alchimie« wurde gezeigt, wie man mit der BASOUT-Routine die Drucker-Ausgabe betreibt. Dort wurden alle wichtigen Routinen bis ins Detail beschrieben.

Listing 5 gibt einen Text zuerst auf dem Drucker und dann auf dem Bildschirm aus. Daran soll außer dem Druckerbetrieb auch gezeigt werden, wie man die Parameterübergabe an STROUT als Makro (Zeilen 230 - 270) definiert und sich somit einen bequemen Ausgabe-Befehl schafft.

4. Unterprogramme

Ohne die Unterprogramm-Befehle JSR und RTS kommt fast kein Maschinenprogramm aus. Es ist allerdings ziemlich unbekannt, daß beide Befehle das Programm stark verlangsamen. Grund genug für uns, JSR und RTS näher zu betrachten:

Trifft der Prozessor auf JSR, schiebt er den aktuellen Programmzähler plus 2 (= Rücksprungadresse - 1) auf den Stack und springt dann zu der Adresse, die hinter JSR steht. Trifft er auf RTS, holt er die Adresse vom Stapel zurück, erhöht sie um 1 und verwendet sie wieder als Programmzähler.

Bemerkenswert ist, daß die Zugriffe auf den Stapel sich in keiner Weise von den Zugriffen über die Befehle PHA und PLA unterscheiden. Daher muß jedesmal der Stapelzeiger neu errechnet werden. Diese vielen Operationen sind schuld daran, daß JSR und RTS so langsam sind.

Da wir das Problem erkannt haben, können wir damit beginnen, unser Wissen anzuwenden.

a) Unterprogrammverschachtelung

Stellen wir uns folgendes Beispiel vor: ein Hauptprogramm ruft das Unterprogramm 1 auf. Dieses ruft an seinem Ende das Unterprogramm 2 auf, um dann mit RTS ins Hauptprogramm zurückzukehren.

Alles ziemlich schwierig, oder?

Deshalb gehen wir mit Hilfe einer Grafik vor: In Bild 1 sehen Sie ein Flußdiagramm nach obigem Aufbau. In der Beschriftung soll »Code« nicht »Kennwort« bedeuten, sondern heißt einfach »Befehlsnummer«.

Wie an den Pfeilen zu erkennen ist, werden zwei RTS-Befehle hintereinander abgearbeitet (von Unterprogramm 2 nach Unterprogramm 1 und von dort zum Hauptprogramm).

```
100  -.LI 1,3,0
110  -;
120  -; TEXT IN VIDEO-RAM SCHREIBEN
130  -;
140  -.BA $C000 ; START: SYS 49152
150  -;
180  -.GL CLRSCR = $E544 ; BILDSCHIRM LOESCHEN
190  -;
200  -.GL ZEILE  = 12
210  -.GL SPALTE = 10
220  -;
230  -.GL VIDEORAM = 1024 ; BILDSCHIRMSPEICHER
240  -.GL ADRESSE = VIDEORAM + (40*ZEILE) + SPALTE
250  -;
255  -           JSR CLRSCR      ; = PRINT CHR$(147)
260  -           LDX #0
270  -SCHLEIFE   LDA TEXT,X      ; BILDSCHIRMCODE LESEN
280  -           BEQ ENDE        ; =0, DANN ENDE
290  -           STA ADRESSE,X   ; IN BILDSCHIRMSPEICHER
295  -           INX
296  -           JMP SCHLEIFE    ; NAECHSTES ZEICHEN
300  -ENDE       RTS
305  -;
310  -TEXT       .BY 4,1,19," ",9,19,20," "
311  -.BY 4,5,18," ",20,5,24,20,"!"
320  -.BY 0 ; ENDMARKIERUNG DES TEXTES
```

Listing 4. Die schnellste Lösung, einen Text auszugeben

```
100  -.LI 1,3,0
110  -;
120  -; DRUCKER-AUSGABE MIT
130  -; DER STROUT-ROUTINE
140  -;
150  -.GL STROUT = $AB1E
160  -.GL SETNAM = $FFBD ; DIE BEDEUTUNG
170  -.GL SETLFS = $FFBA ; DIESER ROUTINEN
180  -.GL OPEN   = $FFC0 ; ENTNEHMEN SIE
190  -.GL CHKOUT = $FFC9 ; BITTE DEM KURS
200  -.GL CLRCHN = $FFCC ; "ASSEMBLER IST
210  -.GL CLOSE  = $FFC3 ; KEINE ALCHIMIE"
220  -;
230  -.MA PRINT (ADRESSE)
240  -           LDA #<(ADRESSE)
250  -           LDY #>(ADRESSE)
260  -           JSR STROUT
270  -.RT
280  -;
290  -.BA $C000 ; START: SYS 49152
300  -;
310  -           LDA #0          ; KEINEN
320  -           JSR SETNAM      ; FILENAMEN
330  -;
340  -           LDA #4          ; LOG. FILENUMMER =4
350  -           TAX             ; GERAETEADRESSE 4
360  -           LDY #0          ; SEKUNDAERADRESSE 0
370  -           JSR SETLFS      ; PARAMETER SETZEN
380  -;
390  -           JSR OPEN        ; FILE OEFFNEN
400  -;
410  -           LDX #4          ; FILENUMMER 4
420  -           JSR CHKOUT      ; AUSGABE AUF DRUCKER LENKEN
430  -;
440  -...PRINT (TEXT) ; TEXT AUSGEBEN
450  -;
460  -           JSR CLRCHN      ; WIEDER BILDSCHIRMAUSGABE
470  -;
480  -...PRINT (TEXT) ; JETZT AUF BILDSCHIRM
490  -;
500  -           LDA #4          ; LOG. FILENUMMER 4
510  -           JMP CLOSE       ; FILE SCHLIESSEN
520  -; & PROGRAMM BEENDEN
530  -;
540  -TEXT       .TX "DIESER TEXT WIRD AUF"
550  -.TX " DEN DRUCKER AUSGEGEBEN !"
560  -.BY 13,13,13,0 ; 3 * CAR.RETURN
```

Listing 5. So gibt man Text auf dem Drucker aus

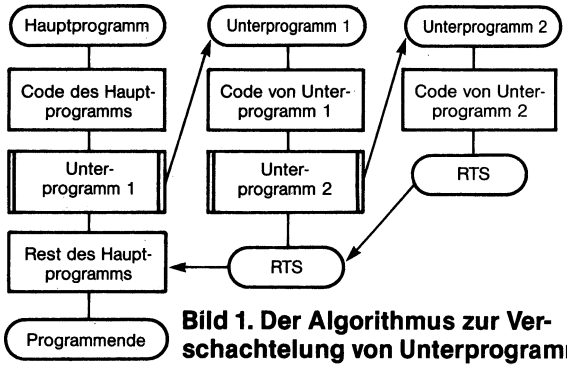

Bild 1. Der Algorithmus zur Verschachtelung von Unterprogrammen

Dies ist immer ein Indiz dafür, daß das Programm noch optimiert werden kann.

Eine »Übersetzung« von Bild 1 in Assembler ist Listing 6: Wenn Sie dieses über »SYS 49152« starten, ist aus den ausgegebenen Texten ersichtlich, welcher Programmteil wann abgearbeitet wird.

Sobald Sie die Struktur von Bild 1 beziehungsweise Listing 6 verstanden haben, können wir uns mit der optimierten Form befassen, die in Bild 2 beziehungsweise Listing 7 zu finden ist.

Hier wird das ehemalige Unterprogramm 2 ans Ende von Unterprogramm 1 gehängt (wobei es ebenfalls über JMP UP2 angesprungen werden könnte). Auf diese Weise muß es nicht über JSR aufgerufen werden, was auch einen RTS-Befehl überflüssig macht.

Trotz dieser Änderung kann das Unterprogramm 2 auch weiterhin als Unterprogramm aufgerufen werden, da bei JSR UP2 die CPU auf einen RTS-Befehl trifft (Bild 2).

In Listing 7 muß noch der JMP-Befehl in Zeile 480 erläutert werden:

Dort muß nicht JSR STROUT:RTS stehen, weil am Ende der STROUT-Routine im ROM ohnehin ein RTS steht. Deshalb benötigt unser Programm keinen eigenen RTS-Befehl zur Rückkehr ins Hauptprogramm.

Die folgende Regel gilt für Aufrufe von Betriebssystemroutinen:

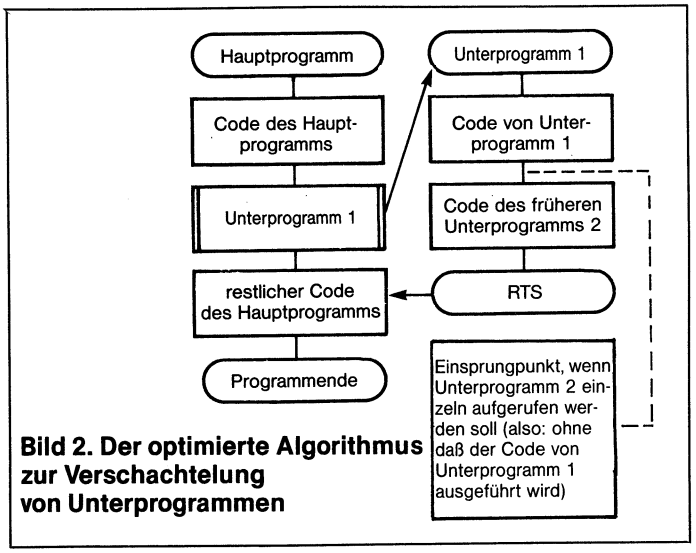

Bild 2. Der optimierte Algorithmus zur Verschachtelung von Unterprogrammen

JSR $XXXX entspricht JMP $XXXX
RTS

Voraussetzung ist, daß im Unterprogramm ab $XXXX keine Stapelmanipulation erfolgt, wie sie gleich beschrieben wird. Das geschilderte Verfahren zur Unterprogrammverschachte-

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -; UNTERPROGRAMMVERSCHACHTELUNG
140  -; (OPTIMIERTE ASSEMBLERVERSION)
150  -;
160  -.GL STROUT = $AB1E
170  -;
180  -.MA PRINT (ADRESSE)
190  -           LDA #<(ADRESSE)
200  -           LDY #>(ADRESSE)
210  -           JSR STROUT
220  -.RT
230  -;
240  -; ---------------- HAUPTPROGRAMM
250  -;
260  -...PRINT (TEXT1)
270  -;
280  -           JSR UP1
290  -;          ↑ AUFRUF VON UNTERPROGRAMM 1
300  -;
310  -...PRINT (TEXT2)
320  -;
330  -           JMP $A474       ; WARMSTART
340  -;
350  -;
360  -; ---------------- UNTERPROGRAMM 1
370  -;
380  -UP1        NOP             ; BELIEBIGER CODE
390  -...PRINT (TEXT3)
400  -;
410  -;
420  -;
430  -; ---------------- CODE VON UNTERPROGRAMM 2
440  -;
450  -UP2        NOP             ; BELIEBIGER CODE
460  -           LDA #<(TEXT4)   ; LOW-BYTE
470  -           LDY #>(TEXT4)   ; HIGH-BYTE
480  -           JMP STROUT      ; TEXTAUSGABE
490  -; UND RUECKSPRUNG VOM UNTERPROGRAMM,
500  -; WEIL AM ENDE DER STROUT-ROUTINE
510  -; EIN RTS-BEFEHL STEHT.
10000-;
10010-;
10020-; ---------------- TEXTE
10030-;
10040-TEXT1      .TX "HIER IST DAS HAUPTPROGRAMM."
10050-.BY 13,13 ; 1 LEERZEILE
10060-.BY 0 ; ENDMARKIERUNG
10070-;
10080-TEXT2      .TX "HIER IST WIEDER DAS HAUPTPROGRAMM."
10090-.BY 13,13,0
10100-;
10110-TEXT3      .TX "HIER IST DAS UNTERPROGRAMM 1."
10120-.BY 13,13,0
10130-;
10140-TEXT4      .TX "HIER IST DAS UNTERPROGRAMM 2."
10150-.BY 13,13,0
```

Listing 6. Die umständliche Methode, Unterroutinen aufzurufen

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -; UNTERPROGRAMMVERSCHACHTELUNG IN ASSEMBLER
140  -;
150  -.GL STROUT = $AB1E
160  -;
170  -.MA PRINT (ADRESSE)
180  -           LDA #<(ADRESSE)
190  -           LDY #>(ADRESSE)
200  -           JSR STROUT
210  -.RT
220  -;
230  -; ---------------- HAUPTPROGRAMM
240  -;
250  -...PRINT (TEXT1)
260  -;
270  -           JSR UP1
280  -;          ↑ AUFRUF VON UNTERPROGRAMM 1
290  -;
300  -...PRINT (TEXT2)
310  -;
320  -           JMP $A474       ; WARMSTART
330  -;
340  -;
350  -; ---------------- UNTERPROGRAMM 1
360  -;
365  -UP1        NOP             ; BELIEBIGER CODE
370  -...PRINT (TEXT3)
380  -;
390  -           JSR UP2
400  -;          ↑ AUFRUF VON UNTERPROGRAMM 2
410  -;
420  -           RTS             ; UP1 VERLASSEN
430  -;
440  -;
450  -; ---------------- UNTERPROGRAMM 2
460  -;
465  -UP2        NOP             ; BELIEBIGER CODE
470  -...PRINT (TEXT4)
480  -;
490  -           RTS             ; UP2 VERLASSEN
500  -;
10000-;
10010-; ---------------- TEXTE
10020-;
10030-TEXT1      .TX "HIER IST DAS HAUPTPROGRAMM."
10040-.BY 13,13 ; 1 LEERZEILE
10050-.BY 0 ; ENDMARKIERUNG
10060-;
10070-TEXT2      .TX "HIER IST WIEDER DAS HAUPTPROGRAMM."
10080-.BY 13,13,0
10090-;
10100-TEXT3      .TX "HIER IST DAS UNTERPROGRAMM 1."
10110-.BY 13,13,0
10120-;
10130-TEXT4      .TX "HIER IST DAS UNTERPROGRAMM 2."
10140-.BY 13,13,0
```

Listing 7. Die optimierte Methode, Unterroutinen aufzurufen

lung und die entsprechenden Regeln können Sie dann auf jede (!) Programmiersprache übertragen.

b) Stapelmanipulation

Wenn Sie »Exbasic Level II« kennen, wissen Sie sicher den Befehl »DISPOSE RETURN« zu schätzen. Er dient dazu, ein Unterprogramm ohne RETURN abzuschließen. Dadurch kann dieses zum Beispiel über GOTO verlassen werden.

In Assembler ist dies auch möglich. Die Befehlseingabe

```
PLA
PLA
```

entspricht in der Wirkung »DISPOSE RETURN«.

Da die Rücksprungadresse auf den Stapel abgelegt wird und dort 2 Byte in Anspruch nimmt, kann sie über PLA:PLA wieder vom Stapel geholt werden. Ein Unterprogramm ist nach PLA:PLA eigentlich kein Unterprogramm mehr, sondern Bestandteil des aufrufenden Programms. PLA:PLA findet vor allem in der Fehlerbehandlung Anwendung. An einem späteren Listing werden wir dies noch sehen. Nach PLA:PLA kann ein Unterprogramm über JMP verlassen werden. Dies machen wir uns zunutze, um den Rücksprung an eine beliebige Adresse zu simulieren. Dies ist sonst nicht möglich, da bei RTS immer hinter den Befehl gesprungen wird, der das Unterprogramm aufgerufen hat.

Ein RTS an eine beliebige Adresse müßte »RTS XXXX« heißen, doch diesen Befehl gibt es beim 6510 nicht. So wird er aber simuliert:

```
PLA          ; holt Rücksprungadresse
PLA          ; vom Stapel und
JMP $XXXX    ; springt nach $XXXX
```

So sieht ein Makro dazu aus:

```
-.MA    RTS (RUECKSPRUNGADRESSE)
-       PLA
-       PLA
-       JMP RUECKSPRUNGADRESSE
-.RT
```

Und noch ein Mangel der Unterprogrammbefehle soll beseitigt werden: Obwohl es JMP (indirekt) gibt, kennt der 6510 keinen Befehl wie JSR (indirekt); über Stapelmanipulation ist dies dennoch möglich (siehe dazu auch im 64'er, Ausgabe 1/86: Assembler-Bedienung leicht gemacht).

Nehmen wir an, im Vektor $14/$15 steht die Adresse $C000. Nun soll über den $14/$15-Vektor ein Unterprogramm aufgerufen werden (also das ab $C000). Bild 3 zeigt, was im einzelnen geschehen muß.

Die Rücksprungadresse steht zwar in Bild 3 direkt hinter dem JMP ($0014)-Befehl, kann aber auch anderswo im Programm stehen.

Folgendes Makro ermöglicht die Simulation von JSR (indirekt):

```
-. MA JSRIND (VEKTOR, RUECKSPRUNGADRESSE)
-       LDA # >(RUECKSPRUNGADRESSE-1)
-       PHA
-       LDA # <(RUECKSPRUNGADRESSE-1)
-       PHA
-       JMP (VEKTOR)
-. RT
```

Diese Simulation von JSR ($XXXX) verwendet auch der SYS-Befehl (disassemblieren Sie von $E12A bis $E155 und betrachten Sie dazu Bild 3).

Zuerst holt er die Zahl nach SYS in die Adressen $14/$15, dann legt er die Rücksprungadresse ($E147) -1 auf dem Stack ab. Nun holt er die Register P, A, X, Y aus den Adressen $030F, $030C, $030D, $030E. Es folgt ein indirekter Sprung über $0014/$0015.

Nach dem Rücksprung werden die Register wieder im Speicher dort abgelegt, woher sie genommen wurden und ein Sprung ins Basic wird durchgeführt.

Später werden wir noch eine weitere Möglichkeit für JSR (ind) kennenlernen, die aber nicht auf Stapelmanipulation beruht.

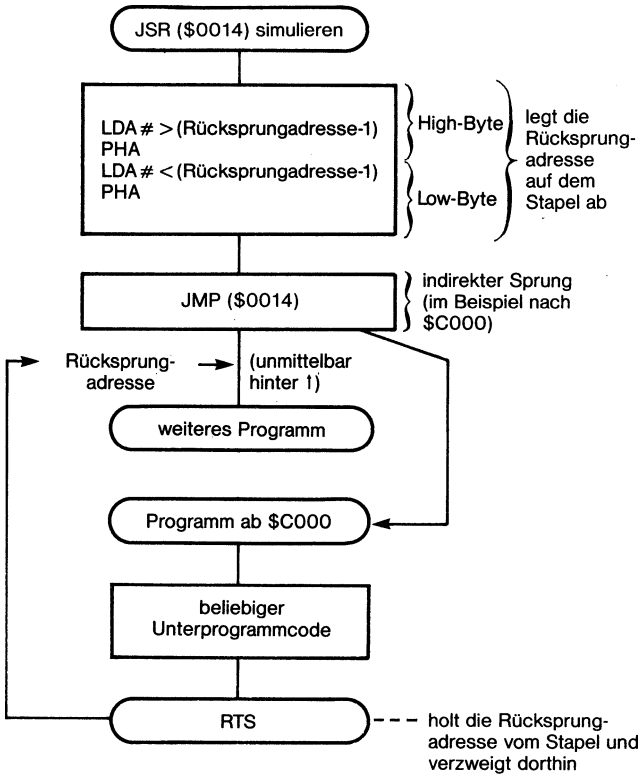

Bild 3. Der Algorithmus, um einen JSR (indirekt)-Befehl zu simulieren

c) Vergleich zwischen Unterprogramm und Makro bezüglich Geschwindigkeit

Wenn Sie den Hypra-Ass (oder einen anderen Makro-Assembler) besitzen, haben Sie die Möglichkeit, Befehlsfolgen als Makros zu definieren. Makros sind deswegen so beliebt, weil sie den größten Vorteil von Unterprogrammen bieten, nämlich Übersichtlichkeit. Da Makros aber wie »normale« Befehle im Speicher stehen, entfällt der Aufruf über JSR und RTS. Dies ist der Grund, weshalb Makros etwas schneller (wenige Taktzyklen) als Unterprogramme sind. Das Problem, wann Makros und wann Unterprogramme vorteilhaft sind, wird später noch aufgegriffen.

5. Tabellen

Im allgemeinen Sprachgebrauch werden Tabellen als »geordnete Zusammenstellungen von Daten« verstanden. Diese Funktion haben sie auch in Computerprogrammen, wo man sie daran erkennt, daß Tabellen keinen Befehlscharakter haben.

SMON-Benutzer können mit »FT« ein Programm nach Tabellen durchsuchen lassen; dann sucht SMON im Programm nach Bytes, die nicht zu Maschinensprachebefehlen gehören.

Wozu werden nun Tabellen verwendet?

In der Regel dienen Tabellen einem Computerprogramm als »elektronischer Rechenschieber«. So wie das Kopfrechnen durch einen Rechenschieber ersetzt werden kann, weil man nur in einer geordneten Zusammenstellung von Ergebnissen das richtige suchen muß, kann ein Programm aus seinen Tabellen denselben Nutzen ziehen: die Berechnungen entfallen, die Programmierung wird einfacher.

Aus den weniger erforderlichen Berechnungen entsteht ein deutlicher Geschwindigkeitszuwachs, der Hauptvorteil von Tabellen. Wie man Tabellen einsetzt, erfahren Sie im folgenden.

a) Tabellen aus Rechenergebnissen

Noch einmal zum Rechenschieber. Es geht beim Kopfrechnen viel schneller, 4x10 auszurechnen als 4x7. Bei einem Rechenschieber besteht kaum ein Unterschied in der »Rechenzeit«.

Dementsprechend existiert fast kein Algorithmus, dessen Ausführungszeit bei unterschiedlichen Parametern immer gleich bliebe. Wer den Artikel »Dem Klang auf der Spur (5)« (64'er, Ausgabe 5/85, Seite 152 ff.) gelesen hat, weiß, welch grobe Differenzen bei Multiplikationen auftreten können.

Ersetzt (beziehungsweise unterstützt) man einen Algorithmus durch eine Multiplikationstabelle, fällt eine einheitlichere (und kürzere) Ausführungszeit an.

Für das Rechnen mit einzelnen Bits in einem Byte werden oft die Zweierpotenzen benötigt; es empfiehlt sich, diese als Tabelle anzulegen:

```
1000 -; Zweierpotenzen als Tabellle
1010 -; im DOS der Floppy 1541 ab $EFE9
1020 -; zu finden
1030 -; ZWEIPOT .BY 2↑0, 2↑1, 2↑2, 2↑3, 2↑4,
        2↑5, 2↑6, 2↑7
```

Folgende Unterroutine legt im Akkumulator den Wert 2↑A ab, wobei mit A der Inhalt des Akkumulators bei Aufruf der Routine gemeint ist:

```
10000 -;
10010 -; Subroutine zur Berechnung von
10020 -; 21A (Ergebnis kommt in den Akku)
10030 -;
10040 - TAX ; Akku in Indexregister
10050 - LDA ZWEIPOT,X ; aus Tabelle einlesen
10060 - RTS ; Das war's schon! Wer ein
        schnelleres und zugleich so einfaches
        Verfahren kennt, möge sich melden...
10070 - ZWEIPOT
        .BY 2↑0,2↑1,2↑2,2↑3,2↑4,2↑5,2↑6,2↑7
```

Wenn A größer als 7 ist, liefert das Programm falsche Werte. Sie können es noch erweitern, wenn Sie es für nötig halten.

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -; RECHNUNG MIT FLIESSKOMMAWERTEN
140  -;
150  -.GL MEMFAC = $BBA2
160  -.GL FACOUT = $AABC
170  -.GL SQRFAC = $BF71
180  -.GL LOGNAT = $B9EA
190  -;
200  -.MA HOLE (ADRESSE) ; MAKRO-DEF.
210  -          LDA #<(ADRESSE); HOLT MFLPT-ZAHL
220  -          LDY #>(ADRESSE); VON ADRESSE IN
230  -          JSR MEMFAC     ; DEN FAC
240  -.RT
250  -;
260  -;
270  -...HOLE (BSPZAHL)
280  -;
290  -          JSR FACOUT     ; AUSDRUCKEN
300  -;
310  -...HOLE (BSPZAHL)
320  -;
330  -          JSR SQRFAC     ; QUADRATWURZEL
340  -;
350  -          JSR FACOUT     ; AUSDRUCKEN
360  -;
370  -...HOLE (BSPZAHL)
380  -;
390  -          JSR LOGNAT     ; LOGARITHMUS NATURALIS
400  -;
410  -          JMP FACOUT     ; AUSDRUCKEN
500  -;
510  -; BEISPIELZAHL 1.23456
520  -; IM MFLPT-FORMAT
530  -;
540  -BSPZAHL   .BY $81,$1E,$06,$0F,$E5
550  -;
```

Listing 8. Fließkommazahlen in Assembler verarbeiten

b) Tabellen aus Fließkommawerten

Zu den zeitraubendsten Operationen gehört die Rechnung mit Fließkommazahlen. Daß diese selbst in Maschinenprogrammen lähmend wirkt, sehen Sie am HiRes-3-Befehl »FUNKT« (64'er, Ausgabe 3/85, Grafikkurs-Anwendung). Daher sollte man nur dann auf die Fließkommaroutinen zugreifen, wenn es unvermeidbar ist. Berechnen Sie soviele Werte wie möglich voraus, hierfür eignet sich der Direktmodus des Basic-Interpreters besonders gut! Wie Sie einen auf diese Weise berechneten Wert ins MFLPT-(Floating Point)Format umwandeln können, zeigt Ihnen der folgende Kasten.

Verfahren zur Umwandlung einer Zahl ins MFLPT-Format

1. SMON (oder anderen Monitor) laden
2. RESET auslösen oder NEW eingeben
3. »XX = Fließkommazahl« eingeben, zum Beispiel »XX = 1.23456«
4. Monitor starten (SYS 49152)
5. »M 0805 0809« eingeben

Sie sehen nun in den Adressen $0805 - $0809 die MFLPT-Darstellung der Zahl, mit der Sie die Variable XX belegt haben.

Damit wir uns unter Zuhilfenahme präziser Fachausdrücke und Abkürzungen verständigen können, sollten Sie den Abschnitt in »Assembler ist keine Alchimie« aufmerksam lesen, der sich mit Fließkommazahlen befaßt. Nach dem Studium dieses Abschnitts sollten Ihnen Begriffe wie »MFLPT«, »FAC« oder »ARG« geläufig sein.

Im Falle der Zahl 1.23456 erhalten wir als Ergebnis:

```
:0805 81 1E 06 0F E5...
```

Diese Werte legen wir folgendermaßen als Tabelle ab:

```
540 -BSPZAHL .BY $81, $1E, $06, $0F, $E5
```

Wie wir nun diese Zahl verarbeiten, zeigt Ihnen Listing 8. Das Makro (200 - 240) stützt sich auf die Interpreter-Routine MEMFAC, die eine Zahl (Adresse wird in Akku/Y-Register übergeben) vom Speicherformat MFLPT in den FAC als FLPT-Zahl schreibt und dabei die erforderliche MFLPT-→FLPT-Umwandlung durchführt.

In der Tabelle in Zeile 540 können Sie beliebige Fließkommawerte (sofern Sie diese wie angegeben berechnet haben) einsetzen, das Programm rechnet dann mit der jeweiligen Fließkommazahl, die ab BSPZAHL im MFLPT-Format steht.

Diese Zahl wird zunächst nur in den FAC geladen und der FAC wird dann ausgedruckt (270 - 290), dann wird die Zahl wieder geholt, die Wurzel berechnet und ausgegeben (310 - 350). Schließlich wird die Zahl wieder in den FAC geholt, der natürliche Logarithmus errechnet und auch ausgegeben (370 - 410).

Zur Routine FACOUT sind, außer daß sie den Inhalt des FAC ausgibt, noch zwei Bemerkungen zu machen:

1. Nach der Zahl wird noch ein CARRIAGE RETURN ausgegeben.
2. Nach dem Aufruf von FACOUT hat sich der Inhalt des FAC aufgrund mehrerer Divisionen durch Zehnerpotenzen verändert.

Auf das Thema »Fließkommaarithmetik« geht Texteinschub 1 noch näher ein. Dort werden auch weitere Interpreter-Routinen vorgestellt.

c) Sprungtabelle

Beim Thema »Unterprogramme« wurde Ihnen eine Methode vorgestellt, um JSR (ind) zu simulieren. Diese erweist sich in Verbindung mit einer Tabelle, in der die Sprungadressen gespeichert sind, als sehr nützlich. So kann beispielsweise eine Parallele zum Basic-Befehl ON...GOSUB ZIEL1,ZIEL2.... geschaffen werden.

Ein Beispiel: Wenn der Basic-Interpreter auf einen Basic-Befehl trifft, holt er aus der Tabelle $A00C – $A09D die Adresse der zugehörigen Routine. Diese springt er dann durch Stapelmanipulation an.

Der SMON arbeitet genauso: Seine Sprungtabelle liegt im Bereich $C02B – $C06B.

Die Anwendung von Sprungtabellen werden wir noch ausführlich im folgenden Abschnitt d) sowie bei der Besprechung von Listing 11 behandeln.

d) Vergleichstabellen

Weder der SMON noch der Basic-Interpreter benutzen zum Suchen der zum jeweiligen Befehl gehörenden Routine eine Reihe von CMP-Abfragen mit BRANCH-Befehlen. Auch für die Vergleichswerte (in diesem Fall die Befehlswörter) gibt es eine Tabelle: Beim SMON liegt sie im Bereich $C00B – $C02A, beim Basic-Interpreter $A09E – $A327.

Sprung- und Vergleichstabellen sind in gleicher Befehlsfolge angeordnet; wird der Befehl an einer bestimmten Stelle in der Vergleichstabelle gefunden, erfolgt ein Sprung an die Adresse, die an gleicher Stelle in der Sprungtabelle steht.
So sehen die Befehls- und Vergleichstabellen im SMON aus:

Spalte Nr.	1	2	3	4	....
Befehl	/	#	$	%	....
Sprungadr. $	CADB	C920	C908	C91C	....

Die Sprungadressen sind wegen der Stapelmanipulation in der Tabelle ab $C02B um 1 dekrementiert gespeichert; in der Darstellung sehen Sie aber das tatsächliche Sprungziel.

Wir werden jetzt anhand des SMON die Verwendung einer Vergleichs-Sprungtabelle in Assembler erläutern.

Wenn wir die zum Befehl »#« gehörende Sprungadresse finden wollen, gehen wir folgendermaßen vor:

1. Wir suchen in Reihe 2 das #-Zeichen.
2. Wir gehen (in derselben Spalte) eine Reihe nach unten und finden dort die Sprungadresse ($C92C).

Der Computer hat nicht die Möglichkeit, direkt eine Reihe weiter unten die Suche fortzusetzen. Er muß einen Umweg wählen und sich die Spalte merken. Ein Beispiel:

1. Der SMON sucht unter den Elementen aus Reihe 2 das »#«. In einem Zähler merkt er sich die Spalte, in der der Befehl gefunden wurde.
2. Nun sucht er in Reihe 3 in der Spalte, die im Zähler steht, die zugehörige Sprungadresse.

Wie ähnlich beide Suchvorgänge sind, erkennen Sie daran, daß jedesmal die Hauptschritte 1. und 2. vorkommen.

Nach so viel Theorie sehen wir uns nun umso ausführlicher die Routine im SMON an, die für die Steuerung der Vergleichs-Befehlstabelle verantwortlich ist. Dazu können Sie »D C303 C323« eingeben.

Bei Adresse $C303 steht im Akku der ASCII-Code des Kommandos, das der SMON ausführen soll (zum Beispiel $40, wenn ein M-Befehl eingegeben wurde).

C303 LDX #$20	32-1 Befehle müssen durchsucht werden. Weshalb »-1« erforderlich ist, liegt an der Schleifenstruktur und ist unbedeutend.
C305 CMP $C00A,X	Akku (enthält Befehl) mit X-tem Element der Befehlstabelle vergleichen; $C00A = Befehlstabelle -1, weil Adresse $C00A nie zum Vergleich herangezogen wird.
C308 BEQ $C30F	Vergleich positiv; im X-Register steht jetzt die Spalte.
C30A DEX	Zähler wird dekrementiert; es handelt sich hier um eine »Dekrementierschleife« (dieses Thema wird noch behandelt).
C308 BNE $C305	Wenn der Zähler noch nicht gleich 0 ist, folgt ein Sprung zum Schleifenbeginn.
C30D BEQ $C2D1	Wenn X=0, dann wurde die ganze Tabelle durchsucht, und der Befehl nicht gefunden! Deshalb wird in die SMON-Fehlerbehandlung gesprungen.
C30F JSR $C315	Diese Stelle wird von $C308 aus angesprungen; hier wiederum steht ein Aufruf des Unterprogramms ab $C315, das etwas weiter unten besprochen wird.
C312 JMP $C2D6	Nachdem nun der Befehl durch die Subroutine $C315 abgearbeitet wurde, folgt ein Sprung zur Eingabe des nächsten Befehls.

C315 TXA	Das ist sie, die Subroutine! Weil im X-Register die Nummer des Befehls (= Spalte in Tabelle) steht, kommt das X-Register ins Hauptrechenregister.
C316 ASL	Die Befehlsnummer wird mit 2 multipliziert...
C317 TAX	und kommt wieder ins X-Register. Die Multiplikation mit 2 ist erforderlich, weil in der Sprungtabelle ein Element doppelt so lang ist, wie in der Vergleichstabelle, nämlich 2 Byte. Die Sprungadressen belegen deshalb 2 Byte, weil sie aus Low- und High-Bytes bestehen.
C318 INX	Das X-Register wird um 1 erhöht, da das High-Byte eine Position hinter dem Low-Byte steht.
C319 LDA $C029, X	High-Byte wird gelesen. Die Sprungtabelle beginnt zwar 2 Byte nach $C029, aber weil es keine Spalte 0 gibt, muß der Speicherbedarf einer Sprungadresse (=2) abgezogen werden.
C31C PHA	Das High-Byte der Adresse wird auf den Stapel gelegt.
C31D DEX	-1, weil Low-Byte eine Adresse vor High-Byte steht.
C31E LDA $C029,X	Nun wird auch das Low-Byte der Adresse
C321 PHA	auf den Stapel geschoben.
C322 RTS	Der Befehl RTS wird hier zur Simulation von JMP (ind) verwendet. Auf dieses (unpraktische) Verfahren soll nicht weiter eingegangen werden, weil der 6510 den Befehl JMP (ind) kennt. Wichtig ist für uns nur, daß jede SMON-Routine mit einem RTS abgeschlossen wird, dann erfolgt ein Rücksprung zur Adresse $C312.

Damit haben wir SMONs Schleife zum Suchen eines Befehls und dessen Routine durchleuchtet. Sofern Sie ein ROM-Listing zur Verfügung haben, können Sie sich zusätzlich die entsprechenden Stellen im Basic-Interpreter ansehen. Dieser aber benötigt wegen seiner unterschiedlich langen Befehle einen etwas komplizierteren Suchalgorithmus, was wiederum zu erheblich höherer Ausführungszeit beiträgt.

6. Vergleiche von Prüfsummen

Nun lernen wir ein besonders raffiniertes Vergleichsverfahren kennen:

Wie gesagt, benötigen Vergleiche mit Wörtern, die aus unterschiedlich vielen Zeichen bestehen, mehr Taktzyklen. Dies wäre nicht so, wenn wir alle Zeichen auf eine einheitliche Länge bringen würden. Genau dies tut der Basic-Interpreter: Bei Eingabe einer Zeile wandelt er alle Basic-Befehlswörter in Token um. Jedes Token vertritt einen Befehl und kann, da es nur ein Byte benötigt, schneller erkannt werden, als es bei mehreren Bytes möglich wäre.

Ein Nachteil ist jedoch der Speicherplatzaufwand; für die Umwandlung müssen die Befehle irgendwo im Speicher in Langform vorhanden sein.

Es gibt aber noch ein anderes Verfahren, einer Zeichenkette einen Wert zuzuweisen: Die Prüfsummenberechnung. Diese führen zum Beispiel die Eingabehilfen »Checksummer« und »MSE« durch: Aus 8 Byte Programmcode und 2 Byte Adresse errechnet der MSE eine 1 Byte Prüfsumme.

In Bild 4 sehen Sie einen sehr zuverlässigen Algorithmus zur Berechnung von Prüfsummen (insofern zuverlässig, als er sehr unterschiedliche Prüfsummen ermittelt). Listing 9 stellt ein Hilfsprogramm dar, das zu einer Eingabe die Prüfsumme nach dem Algorithmus aus Bild 4 errechnet.

In Listing 9 ist Ihnen eventuell die Routine NUMOUT nicht bekannt. Daher eine Kurzbeschreibung: NUMOUT gibt eine positive Integerzahl, die im Akkumulator (High-Byte) und im X-Register (Low-Byte) übergeben wird, aus. NUMOUT wird zum Beispiel von der LIST-Routine bei der Ausgabe einer Zeilennummer aufgerufen.

Die Routine BASIN soll ebenfalls erklärt werden, da sie in allen folgenden Programmen verwendet werden wird. Wenn die Routine BASIN zum ersten Mal aufgerufen wird, erwartet das Betriebssystem eine Eingabe (normalerweise von Tastatur), die der Eingabe einer Basic-Zeile entspricht. Nach der Eingabe wird das erste eingegebene Byte in den Akku geladen, jeder weitere Aufruf von BASIN holt das nächste Zeichen in den Akku. Wurden alle Bytes eingelesen, wird im Akku der Wert 13 ($0D, RETURN) übergeben. Danach führt ein weiterer Aufruf von BASIN zu erneuter Eingabe von Tastatur.

Ein großer Vorteil von Prüfsummen ist, daß die Vergleiche mit nur einem Byte, nämlich der Prüfsumme, durchgeführt werden müssen.

Wie man in den Genuß dieses Vorteils kommt, zeigt Listing 10. Wenn Sie den Namen eines Computers (C 64, VC 20, PC 128 oder AMIGA) eingeben, nennt das Programm den in diesem Computer installierten Mikroprozessor. Bei der Eingabe der Computernamen kann man aufgrund der Zeilen 230 und 248 beliebig viele Leerzeichen eingeben. Bei der Errechnung der Prüfsummen mit Listing 9 dürfen allerdings keine eingegeben werden, da Listing 9 diese nicht überliest und somit ein falsches Ergebnis liefern würde.

Der Programmteil, der die Prüfsumme der Eingabe berechnet, ist mit Ausnahmen der Zeilen 230/240 aus Listing 9 übernommen worden. Nach Zeile 450 wird die ermittelte Prüfsumme mit der Tabelle »PRÜFSUMMEN« (Zeile 2060) verglichen.

Bei »WEITER2« (Zeile 620) steht im X-Register die Spalte,

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -.GL BASIN = $FFCF
140  -.GL NUMOUT = $BDCD
150  -.GL STROUT = $AB1E
160  -;
170  -ANFANG    LDA #<(TEXT1)
180  -          LDY #>(TEXT1)
190  -          JSR STROUT
200  -;
210  -          LDX #0
220  -SCHLEIFE1 JSR BASIN
230  -          CMP #13          ; 13 = RETURN
240  -          BEQ WEITER
250  -          STA STORE,X
260  -          INX
270  -          JMP SCHLEIFE1
280  -;
290  -WEITER    STX LAENGE
300  -          LDA #<(TEXT2)
310  -          LDY #>(TEXT2)
320  -          JSR STROUT
330  -          LDA #0
340  -; 0 = AUSGANGSWERT DER PRUEFSUMME
350  -          TAX              ; ZAEHLER = 0
360  -SCHLEIFE2 ROL              ; PRUEFSUMME * 2
370  -          EOR STORE,X
380  -          INX              ; ZAEHLER ERHOEHEN
390  -          CPX LAENGE
400  -          BNE SCHLEIFE2
410  -          CLC
420  -          ADC LAENGE       ; LAENGE ADDIEREN
430  -          TAX              ; PRUEFSUMME
440  -          LDA #0           ; AUSGEBEN
450  -          JSR NUMOUT
460  -          JMP ANFANG       ; NOCH EINMAL
1000 -;
1010 -; TEXTE
1020 -;
1030 -TEXT1     .BY 13
1040 -.TX "----------------------------------------"
1050 -.TX "EINGABE ?  "
1060 -.BY 0
1070 -;
1080 -TEXT2     .BY 13
1090 -.TX "PRUEFSUMME "
1100 -.BY 0
2000 -;
2010 -; ZWISCHENSPEICHER
2020 -;
2030 -LAENGE    .BY 0            ; ZWISCHENSPEICHER
2040 -STORE     .BY 0
2050 -; ↑ AB STORE WIRD DIE EINGABE ABGELEGT
```

Listing 9. Die Berechnung von Prüfsummen

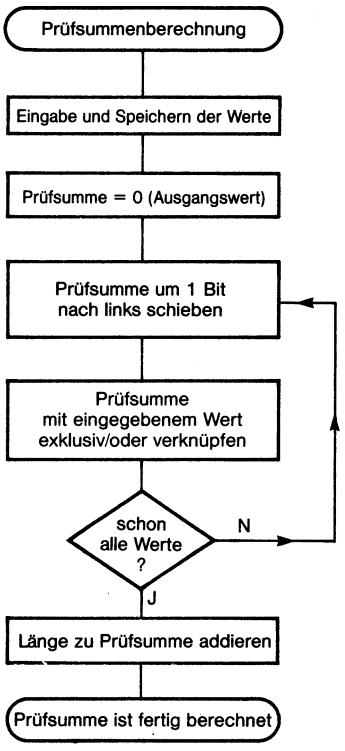

Bild 4. Das Flußdiagramm zur Prüfsummenberechnung

in der die Prüfsumme gefunden wurde. Listing 10 numeriert, im Gegensatz zum SMON die Spalten mit 0 (statt mit 1) beginnend. Außerdem wurde die Adressentabelle in »LOWTAB« (Tabelle der Low-Bytes) und »HIGHTAB« (High-Bytes) zerlegt, was die Programmierung stark erleichtert.

Wir würden zwar Spalten von 1 an numerieren, für den Computer ist es aber besser, mit Spalte 8 zu beginnen. Wenn im X-Register die Spalte (0: VC 20, 1: C 64, 2: PC 128, 3: AMIGA) steht, lesen die Zeilen 620/630 aus einer Tabelle die Adresse, ab der die ASCII-Darstellung des Prozessors zu finden ist. Weil jede der Tabellen »LOWTAB« und »HIGHTAB« gleich viele Elemente wie die Tabelle »PRUEFSUMMEN« hat, muß keine komplizierte Umwandlung über Multiplikation mit 2 oder ähnliches erfolgen wie beispielsweise beim SMON.

Auf eine akute Gefahr bei der Verwendung von Prüfsummen soll jetzt hingewiesen werden: die »Überschneidung von Prüfsummen«:

So wie unterschiedliche Basic-Zeilen beim Checksummer eine gleiche Prüfsumme haben können, sind Prüfsummen nie eindeutig.

Wenn Sie bei Listing 10 etwas herumprobieren, werden Sie sicher feststellen, daß auch eigentlich nicht vorgesehene Eingaben Wirkung zeigen. Dies liegt daran, daß diese Eingaben die gleiche Prüfsumme wie die Taste »VC 20«. »C 64«, »PC 128« oder »AMIGA« haben. Daher sollte man immer darauf achten, daß sich die vorgesehenen Eingaben nicht in ihren Prüfsummen überschneiden (das heißt, die gleichen Prüfsummen haben). Wenn man dies aber beachtet, so ist das Arbeiten mit Prüfsummen, vor allem bei kleineren Datenmengen, eine angenehme Sache.

e) Beispielprogramm für Tabellen

Wenden wir uns jetzt einem etwas größeren (aber keineswegs komplizierteren) Programm zu. Es heißt schlicht und einfach »TABELLEN-BEISPIEL«, womit schon einiges über die Funktion ausgesagt ist: ein reines Beispielprogramm, das nicht den Anspruch erhebt, etwa als Anwendersoftware nützlich zu sein. In Listing 11 finden Sie den kommentierten Quelltext.

Zuerst soll die Bedienung des Programms erläutert werden. Gestartet wird »TABELLEN-BEISPIEL« durch SYS 49152, worauf man sich in folgendem Menü befindet:

ZAHL IN ZAHLWORT WANDELN (0)
BILDSCHIRMFARBE (1)
RESET AUSLOESEN (2)
PROGRAMMENDE UEBER RTS (3)
BITTE AUSWAEHLEN!

Die Zahlen in Klammern sehen Sie nicht, diese zeigen nur die interne Numerierung der Menüpunkte an.

Der jeweils angewählte Menüpunkt (unmittelbar nach dem Start: 0) wird im Gegensatz zu den anderen revers hervorgehoben.

Der angewählte Menüpunkt kommt durch Drücken von F1,RETURN, »←« - oder »=«-Taste zur Ausführung.

Wollen Sie einen anderen Menüpunkt anwählen, drücken Sie einfach CRSR DOWN,»D«,F5 oder »+«, um den invertierten Bereich nach unten zu bewegen. Weiter nach oben gelangen Sie über CRSR UP,»U«,F3 oder »-«.

Wenn Sie von »3« aus nach unten wollen, geht es wieder bei »0« los; von »0« nach oben führt auf Punkt »3«.

Auf Punkt »0« (Ausgangseinstellung) kommen Sie über HOME,»0« oder Klammeraffe.

Sicher würden Sie Ihre Programme auch gerne mit einem solch komfortablen Menü aufwerten. Wenn Sie die Beschreibung des Quelltextes gut durchlesen, wird dies keine Schwierigkeiten bereiten.

Nun zu den einzelnen Menüpunkten.

»2« (Reset auslösen) springt in die RESET-Routine ab $FCE2. »3« (Programmende über RTS) bewirkt einen Rücksprung ins Basic. Wenn Sie aber »TABELLEN-BEISPIEL« vom

```
100  -.LI 1,3,0
110  -.BA $C000 ; START: SYS 49152
120  -;
130  -.GL BASIN = $FFCF
140  -.GL NUMOUT = $BDCD
150  -.GL STROUT = $AB1E
160  -;
170  -ANFANG     LDA #<(TEXT1)
180  -           LDY #>(TEXT1)
190  -           JSR STROUT
200  -;
210  -           LDX #0
220  -SCHLEIFE1 JSR BASIN
230  -           CMP #" "          ; SPACE?
240  -           BEQ SCHLEIFE1     ; DANN UEBERLESEN
250  -           CMP #13           ; 13 = RETURN
260  -           BEQ WEITER1
270  -           STA STORE,X
280  -           INX
290  -           JMP SCHLEIFE1
300  -;
310  -WEITER1    STX LAENGE
320  -           LDA #<(TEXT2)
330  -           LDY #>(TEXT2)
340  -           JSR STROUT
350  -           LDA #0
360  -; 0 = AUSGANGSWERT DER PRUEFSUMME
370  -           TAX               ; ZAEHLER = 0
380  -SCHLEIFE2 ROL               ; PRUEFSUMME * 2
390  -           EOR STORE,X
400  -           INX               ; ZAEHLER ERHOEHEN
410  -           CPX LAENGE
420  -           BNE SCHLEIFE2
430  -           CLC
440  -           ADC LAENGE        ; LAENGE ADDIEREN
450  -; HIER STEHT DIE PRUEFSUMME IM AKKU
460  -;
470  -           LDX #0
480  -SCHLEIFE3 CMP PRUEFSUMMEN,X
490  -           BEQ WEITER2
500  -           INX
510  -           CPX #4
520  -           BNE SCHLEIFE3
530  -; PRUEFSUMME NICHT GEFUNDEN
540  -;
550  -           PLA
560  -           PLA
570  -           LDA #<(TEXT3)
580  -           LDY #>(TEXT3)
590  -           JSR STROUT
600  -           JSR ANFANG        ; VON VORNE
610  -;
620  -WEITER2    LDA LOWTAB,X      ; LOW-BYTE
630  -           LDY HIGHTAB,X     ; HIGH-BYTE
640  -           JSR STROUT
650  -           JMP ANFANG        ; NOCH EINMAL!
660  -;
1000 -;
1010 -; TEXTE
1020 -;
1030 -TEXT1      .BY 13
1040 -.TX "----------------------------------------"
1050 -.TX "COMPUTER : "
1060 -.BY 0
1070 -;
1080 -TEXT2      .BY 13
1090 -.TX "PROZESSOR: "
1100 -.BY 0
1110 -;
1120 -TEXT3      .TX "WEISS ICH NICHT!"
1130 -.BY 0
1140 -;
1150 -;
1160 -T6502      .TX "MOS 6502"
1170 -.BY 0
1180 -;
1190 -T6510      .TX "MOS 6510"
1200 -.BY 0
1210 -;
1220 -T8502      .TX "MOS 8502 & Z80"
1230 -.BY 0
1240 -;
1250 -T68000     .TX "MOTOROLA 68000"
1260 -.BY 0
1270 -;
2000 -;
2010 -; NUMERISCHE TABELLEN
2020 -;
2030 -LOWTAB     .BY <(T6502),<(T6510),<(T8502),<(T68000)
2040 -HIGHTAB    .BY >(T6502),>(T6510),>(T8502),>(T68000)
2050 -;
2060 -PRUEFSUMMEN .BY 228,83,149,136
2070 -; REIHENFOLGE: VC20,C64,PC128,AMIGA
3000 -;
3010 -; ZWISCHENSPEICHER
3020 -;
3030 -LAENGE     .BY 0             ; ZWISCHENSPEICHER
3040 -STORE      .BY 0
3050 -; ↑ AB STORE WIRD DIE EINGABE ABGELEGT
```

Listing 10. Eine Anwendung der Prüfsummenberechnung

Hypra-Ass aus gestartet haben, finden Sie sich im »AUTO-NUMBER«-Modus wieder. Dies ist weder ein Fehler von »TABELLEN-BEISPIEL« noch von Hypra-Ass, sondern liegt daran, daß beide Programme eine bestimmte Adresse verwenden, die Hypra-Ass dann als Aufforderung zur automatischen Zeilennumerierung wertet. Am besten starten Sie »TABELLEN-BEISPIEL« nur vom normalen Basic aus.

Punkt »0« bittet Sie um Eingabe einer Zahl von 0 bis 9 und gibt zur eingegebenen Zahl das Zahlwort aus. Beispiel: Eingabe »0«, Ausgabe »NULL«.

Danach müssen Sie eine Taste drücken, um ins Hauptmenü zu kommen.

Punkt »1« schließlich bietet die Möglichkeit, die Hintergrundfarbe besonders elegant einzustellen: Sie geben einfach die Farbe als Wort ein, zum Beispiel SCHWARZ.

Folgende Eingaben sind vorgesehen:
SCHWARZ,WEISS,ROT,TUERKIS,VIOLETT,GRUEN,BLAU,
GELB,ORANGE,BRAUN,HELLROT,GRAU 1,GRAU 2,
HELLGRUEN,HELLBLAU,GRAU 3

Aufgrund der Überschneidung von Prüfsummen zeigen jedoch auch andere Eingaben Wirkung, zum Beispiel:
SCH,HYPRA ASS,PRINT,COMPUTER-GRAPHIK,
TAGESSCHAU

Nun wollen wir uns mit dem Quelltext befassen.

Ab Zeile 10000 finden Sie die Tabellen. Und weil unser Programm ein Beispiel für die Verwendung von Tabellen sein soll, sind es derer recht viele. Die wichtigsten davon sind jedoch analog der internen Numerierung der Menüpunkte aufgebaut, da sie Daten für die Menüsteuerung beinhalten. Diese Tabellen sind auch mit 0 - 3 numeriert und grafisch in Bild 6 dargestellt.

Sehen wir uns wieder den Quelltext, beginnend mit der ersten Zeile, an.

Auf die Symboldefinitionen (210 - 260) folgt die Initialisierung der Hauptschleife (280 - 310). Diese Initialisierung löscht Bildschirm (280) und Tastaturpuffer (290 - 300). Außerdem wird der aktuelle (= derzeit invers dargestellte) Menüpunkt (immer in der Adresse »MPT« enthalten) auf 0 gesetzt (310). Zeile 310 ist also dafür verantwortlich, daß nach dem Start über SYS 49152 das Inversfeld ganz oben steht (auf Punkt 0).

Die Texte, die der Beschreibung der Menüpunkte dienen, werden in der Hauptschleife »HSCHLEIFE« (350 - 550) ausgegeben. Mit dieser wollen wir uns nun eingehend auseinandersetzen.

Zunächst wird die Tabelle »RVSTAB« gelöscht (350 - 400). Diese Tabelle enthält die Information, ob der erläuternde Text zu einem Menüpunkt invers ausgegeben wird. Wenn nein, so enthält das entsprechende Byte eine »0«, andernfalls eine »18« (= REVERS-ON-Code für Betriebssystem). Das entsprechende Byte aus »RVSTAB« braucht nur vor dem Menüpunkt-Text ausgegeben werden (470 - 480). Die Zeilen 410 - 430 sorgen dafür, daß das Byte in »RVSTAB«, welches sich auf den aktuellen Menüpunkt bezieht, den RVS-ON-Code erhält.

In der Hauptschleife muß das X-Register in »XSAVE« gesichert werden, weil die Routine »STROUT« den Inhalt des X-Registers ändert.

Mit »TASTE« (610) beginnt dann die Tastaturabfrage im Menü. Die Routine »GET« holt ein Zeichen von der Tastatur als ASCII-Code in den Akku. Wurde keine Taste gedrückt, erhält der Akku den Code 0. In diesem Fall wartet 620 auf eine neue Eingabe. Beachten Sie bitte, daß der Akku nach der Zeile 620 NIE den Wert 0 haben kann (dies wird sich bald als nützlich erweisen)!

Wurde nun eine Taste gedrückt, sucht »SCHLEIFE« (630 - 680) in der Tabelle »TASTEN«, die im Quelltext ab Zeile 10210 steht, nach dem eingegebenen Zeichen (wird es nicht gefunden, erfolgt in 690 der Sprung zur neuen Eingabe).

```
100  -.BA $C000 ; START: SYS 49152
110  -;
120  -; ************************
130  -; *                      *
140  -; *  TABELLEN - BEISPIEL  *
150  -; *  ==================  *
160  -; *                      *
170  -; *  BY  FLORIAN MUELLER  *
180  -; *                      *
190  -; ************************
200  -;
210  -.GL STROUT = $AB1E
220  -.GL CURSORHOME = $E566
230  -.GL GET = $FFE4
240  -.GL BASIN = $FFCF
250  -.GL BASOUT = $FFD2
260  -.GL RESET = $FCE2 ; SOFTWARE-RESET
270  -;
280  -START     JSR $E544       ; = PRINT CHR$(147)
290  -          LDA #0          ; TASTATURPUFFER
300  -          STA 198         ; LOESCHEN
310  -          STA MPT
320  -; ↑ SETZT AKTUELLEN MENUEPUNKT AUF 0
330  -HSCHLEIFE JSR CURSORHOME
340  -; ↑ HSCHLEIFE = HAUPTSCHLEIFE
350  -          LDA #0
360  -          TAX
370  -SCHLEIFE1 STA RVSTAB,X
380  -          INX
390  -          CPX #4
400  -          BNE SCHLEIFE1
410  -          LDX MPT
420  -          LDA #18         ; 18 = REVERS EIN
430  -          STA RVSTAB,X
440  -          LDX #0
450  -; ↑ SCHLEIFENZAEHLER INITIALISIEREN
460  -SCHLEIFE2 STX XSAVE       ; X RETTEN
470  -          LDA RVSTAB,X
480  -          JSR BASOUT
490  -          LDA TEXTLO,X    ; ERKLAERUNG
500  -          LDY TEXTHI,X    ; ZUM MENUEPUNKT
510  -          JSR STROUT      ; AUSGEBEN
520  -          LDX XSAVE       ; X WIEDER HOLEN
530  -          INX
540  -          CPX #4
550  -          BNE SCHLEIFE2
560  -;
570  -;
580  -; HIER IST DAS MENUE BEREITS AUF
590  -; DEN BILDSCHIRM AUSGEGEBEN WORDEN.
600  -;
610  -TASTE     JSR GET         ; TASTATURABFRAGE
620  -          BEQ TASTE       ; WARTEN AUF TASTENDRUCK
630  -          LDX #0
640  -SCHLEIFE3 CMP TASTEN,X
650  -          BEQ WEITER1
660  -          INX
670  -          CPX #16
680  -          BNE SCHLEIFE3
690  -          JMP TASTE
700  -WEITER1   TXA
710  -          LSR             ; DIVIDIERT AKKU-
720  -          LSR             ; MULATOR DURCH 4
730  -          TAX
740  -          LDA SP1LO,X
750  -          STA SPRUNG
760  -          LDA SP1HI,X
770  -          STA SPRUNG+1
780  -;
790  -.EQ RUECKSPRUNG = HSCHLEIFE-1
800  -; ↑ LEGT RUECKSPRUNGADRESSE DES
810  -;   UNTERPROGRAMMS FEST.
820  -;
830  -          LDA #>(RUECKSPRUNG)
840  -          PHA
850  -          LDA #<(RUECKSPRUNG)
860  -          PHA
870  -          JMP (SPRUNG)
880  -;
890  -;
900  -HOME      LDX #0
910  -          STX MPT
920  -ENDE      RTS             ; ENDE DES UNTERPRG
930  -;
940  -DOWN      LDX MPT         ; MENUEPUNKT
950  -          INX             ; UM 1 ERHOEHEN
960  -          CPX #4          ; GROESSER ALS 3?
970  -          BEQ HOME        ; DANN =0
980  -          STX MPT         ; SONST UEBERNEHMEN
990  -          RTS             ; ZUR HAUPTSCHLEIFE
1000 -;
1010 -UP        LDX MPT         ; MENUEPUNKT
1020 -          DEX             ; DEKREMENTIEREN
1030 -          BPL ENDUP       ; > 0?
1040 -          LDX #3          ; NEIN, DANN =3
1050 -ENDUP     STX MPT         ; UND UEBERNEHMEN
1060 -          RTS             ; ZUR HAUPTSCHLEIFE
1070 -;
1080 -;
1090 -EXEC      PLA             ; STAPELMANIPULATION
1100 -          PLA
1110 -          LDX MPT
```

Listing 11. »Tabellen-Beispiel«, ein Beispiel zur Verwendung von Tabellen

```
1120 -           LDA SP2LO,X
1130 -           STA SPRUNG
1140 -           LDA SP2HI,X
1150 -           STA SPRUNG+1
1160 -           JMP (SPRUNG)
1170 -;
1180 -;
1190 -;
1200 -ZAHLWORT   LDA #<(TZAHL)   ; AUFFORDERUNG
1210 -           LDY #>(TZAHL)   ; ZUR EINGABE
1220 -           JSR STROUT      ; AUSGEBEN
1230 -           JSR BASIN       ; HOLT ZEICHEN
1240 -           SEC             ; IN BINAERZAHL
1250 -           SBC #"0"        ; UMWANDELN
1260 -           TAX             ; INS X-REGISTER
1270 -;
1280 -; JETZT STEHT IM X-REGISTER
1290 -; DIE EINGEGEBENE ZAHL
1300 -;
1310 -           CMP #10         ; > 10?
1320 -           BCC ZAHLWORT1   ; NEIN=> WEITER
1330 -           JMP ZAHLWORT    ; NEUEINGABE
1340 -;
1350 -ZAHLWORT1 STX XSAVE        ; X RETTEN
1360 -           LDA #<(TWORT)   ; AUFFORDERUNG
1370 -           LDY #>(TWORT)   ; ZUR EINGABE
1380 -           JSR STROUT      ; AUSGEBEN
1390 -           LDX XSAVE       ; X WIEDER HOLEN
1400 -           LDA ZWLO,X      ; ADRESSE DES
1410 -           LDY ZWHI,X      ; ZAHLWORTES HOLEN
1420 -           JSR STROUT      ; UND Z.WORT DRUCKEN
1430 -;
1440 -WAIT       JSR GET         ; WARTET AUF
1450 -           BEQ WAIT        ; TASTENDRUCK
1460 -           JMP START       ; ZUM HAUPTMENUE
1470 -;
1480 -;
1490 -;
1500 -FARBE      LDA #<(TFARBE)
1510 -           LDY #>(TFARBE)
1520 -           JSR STROUT
1530 -           LDX #0
1540 -FARBE1     JSR BASIN       ; HOLT EINGABE
1550 -           CMP #" "        ; SPACE ?
1560 -           BEQ FARBE1      ; JA=>UEBERLESEN
1570 -           CMP #13         ; ENDE DER EINGABE?
1580 -           BEQ FARBE2      ; JA, DANN WEITER
1590 -           STA FARBWORT,X  ; EINGABE SPEICHERN
1600 -           INX             ; ZAEHLER ERHOEHEN
1610 -           JMP FARBE1      ; ZUR SCHLEIFE
1620 -FARBE2     STX 2           ; LAENGE MERKEN
1630 -           LDX #0
1640 -           TXA
1650 -FARBE3     ROL
1660 -           EOR FARBWORT,X
1670 -           INX
1680 -           CPX 2           ; SCHON FERTIG?
1690 -           BNE FARBE3      ; NEIN,ZUR SCHLEIFE
1700 -           CLC             ; LAENGE
1710 -           ADC 2           ; ADDIEREN
1720 -;
1730 -; HIER STEHT IM AKKU DIE PRUEFSUMME
1740 -;
1750 -           LDX #0
1760 -FARBE4     CMP PRUEFSUMMEN,X
1770 -           BEQ FARBE5      ; GEFUNDEN
1780 -           INX
1790 -           CPX #16
1800 -           BNE FARBE4
1810 -           JMP FARBE       ; NEUE EINGABE
1820 -FARBE5     STX 53280       ; BILDSCHIRM-
1830 -           STX 53281       ; FARBE SETZEN
1840 -           JMP START       ; ZUM MENUE
1850 -;
10000-;
10010-; TABELLEN
10020-; ========
10030-;
10040-; TEXTE:
10050-;
10060-PUNKT0     .TX "ZAHL IN ZAHLWORT UMWANDELN"
10070-.BY 13,13,0
10080-;
10090-PUNKT1     .TX "BILDSCHIRMFARBE"
10100-.BY 13,13,0
10110-;
10120-PUNKT2     .TX "RESET AUSLOESEN"
10130-.BY 13,13,0
10140-;
10150-PUNKT3     .TX "PROGRAMMENDE UEBER RTS"
10160-.BY 13,13,13
10170-.TX "BITTE AUSWAEHLEN !"
10180-.BY 0
10190-;
10200-;
10210-TASTEN     .BY 133,13,"←","="; 133=F1,13=RETURN
10220-.BY 19,"0","@",0 ; 19=HOME,0=DUMMY
10230-.BY 17,"D",135,"+" ; 17=CRSR DOWN,135=F5
10240-.BY 145,"U",134,"-" ; 145=CRSR UP,134=F3
10250-;
10260-;
10270-TZAHL      .BY 147         ; CLEAR HOME
10280-.TX "ZAHL (0-9) ? "
10290-.BY 0
10300-;
10310-TWORT      .TX " IN WORTEN : "
10320-.BY 0
10330-;
10340-;
10350-; ZAHLWOERTER (0-9)
10360-;
10370-;
10380-NULL       .TX "NULL"
10390-.BY 0
10400-;
10410-EINS       .TX "EINS"
10420-.BY 0
10430-;
10440-ZWEI       .TX "ZWEI"
10450-.BY 0
10460-;
10470-DREI       .TX "DREI"
10480-.BY 0
10490-;
10500-VIER       .TX "VIER"
10510-.BY 0
10520-;
10530-FUENF      .TX "FUENF"
10540-.BY 0
10550-;
10560-SECHS      .TX "SECHS"
10570-.BY 0
10580-;
10590-SIEBEN     .TX "SIEBEN"
10600-.BY 0
10610-;
10620-ACHT       .TX "ACHT"
10630-.BY 0
10640-;
10650-NEUN       .TX "NEUN"
10660-.BY 0
10670-;
10680-;
10690-TFARBE     .BY 147         ; CLEAR HOME
10700-.TX "WELCHE FARBE ? "
10710-.BY 0
10720-;
10730-;
10740-RVSTAB     .BY 0,0,0,0     ; 4 BYTES RESERVIEREN
10750-;
10760-;
10770-; ZAHLEN:
10780-;
10790-; ADRESSEN DER TEXTE, DIE DIE
10800-; MENUEPUNKTE BESCHREIBEN
10810-;
10820-TEXTLO     .BY <(PUNKT0),<(PUNKT1)
10830-.BY <(PUNKT2),<(PUNKT3)
10840-;
10850-TEXTHI     .BY >(PUNKT0),>(PUNKT1)
10860-.BY >(PUNKT2),>(PUNKT3)
10870-;
10880-;
10890-; ADRESSEN DER ZAHLWOERTER
10900-;
10910-ZWLO       .BY <(NULL),<(EINS),<(ZWEI),<(DREI)
10920-.BY <(VIER),<(FUENF),<(SECHS),<(SIEBEN)
10930-.BY <(ACHT),<(NEUN)
10940-;
10950-ZWHI       .BY >(NULL),>(EINS),>(ZWEI),>(DREI)
10960-.BY >(VIER),>(FUENF),>(SECHS),>(SIEBEN)
10970-.BY >(ACHT),>(NEUN)
10980-;
10990-;
11000-; ADRESSEN DER UNTERROUTINEN
11010-; FUER DIE MENUESTEUERUNG
11020-;
11030-SP1LO      .BY <(EXEC),<(HOME),<(DOWN),<(UP)
11040-;
11050-SP1HI      .BY >(EXEC),>(HOME),>(DOWN),>(UP)
11060-;
11070-;
11080-; ADRESSEN DER EINZELNEN
11090-; MENUEPUNKTE
11100-;
11110-SP2LO      .BY <(ZAHLWORT),<(FARBE)
11120-.BY <(RESET),<(ENDE) ; BEI ENDE STEHT
11130-SP2HI      .BY >(ZAHLWORT),>(FARBE)
11140-.BY >(RESET),>(ENDE) ; EIN RTS-BEFEHL
11150-;
11160-; PRUEFSUMMEN DER FARB-WOERTER
11170-;
11180-PRUEFSUMMEN .BY 41,158,137,212,159,101
11190-.BY 3,2,33,69,201,116,113,121,127,114
11200-;
11210-;
11220-; ZWISCHENSPEICHER
11230-;
11240-MPT        .BY 0           ; 1 BYTE RESERVIEREN
11250-XSAVE      .BY 0
11260-SPRUNG     .WO 0           ; 2 BYTES FREIHALTEN
11270-FARBWORT   .BY 0
11280-; ↑ AB 'FARBWORT' WIRD DIE EINGABE
11290-;   DER FARB-BEZEICHNUNG ABGELEGT.

READY.
```

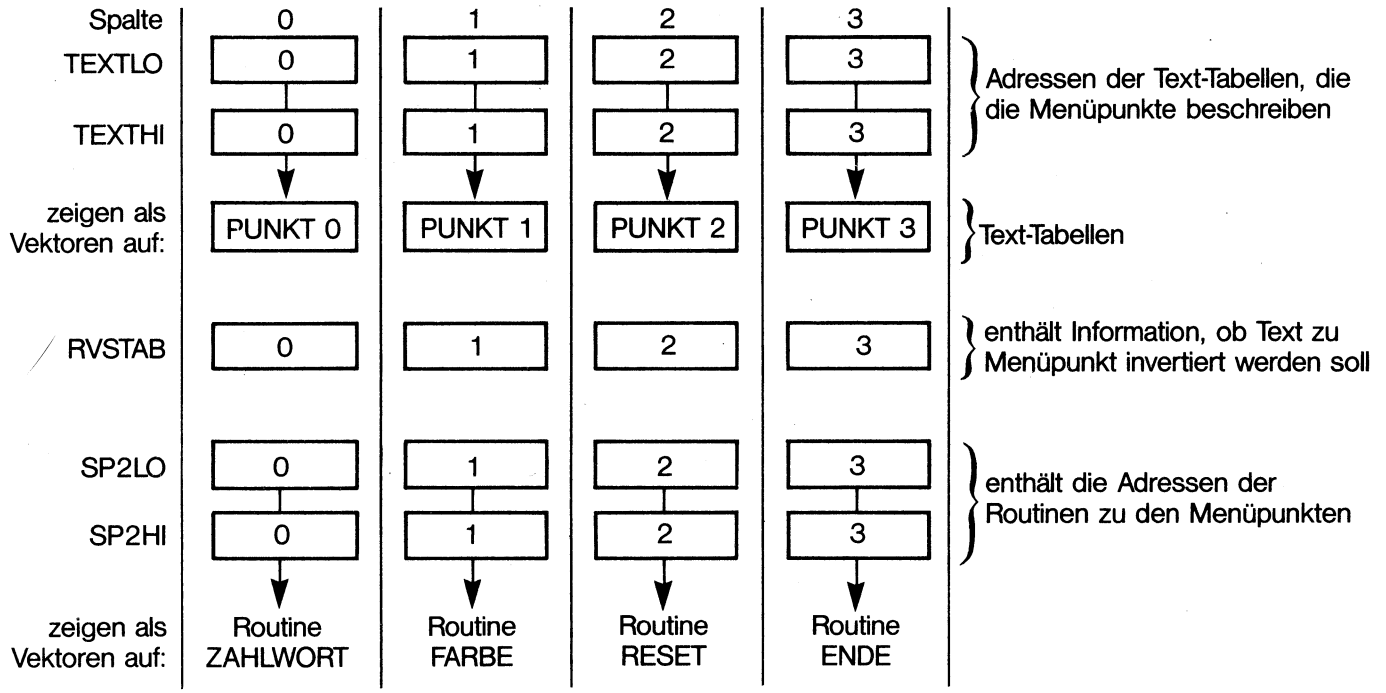

Bild 5. So verwendet man Tabellen zur Realisierung eines Menüs

Diese Tabelle »TASTEN« enthält alle vorgesehenen Tastendrücke zur Menüsteuerung, die in 4er-Blockweise angeordnet sind (Bild 5). Nach der Suchschleife steht im X-Register die Position der gedrückten Taste innerhalb der Tabelle »TASTEN« (zum Beispiel 0 = F1 gedrückt, 4 = HOME gedrückt). Diese Position wird – ohne Berücksichtigung des Divisions-Restes – durch 4 dividiert (700 – 730), um festzuhalten, von welchem Tastenblock eine Taste gedrückt wurde.

Dadurch ist eindeutig bestimmt, welche Befehlsgruppe aufgerufen werden muß.

Steht nach 730 im X-Register 0, wurde eine der ersten vier in »TASTEN« enthaltenen Tasten gedrückt, die die Ausführung des aktuellen Menüpunktes veranlassen (Zeile 10210 und Bild 5). Ist X=1, so wurde eine Taste aus Zeile 10220 gedrückt. In 10220 steht als letztes Byte eine 0. Diese dient, da für die Funktion »Inversfeld in HOME-Position« nur drei Tastendrücke vorgesehen wurden, zum Auffüllen auf vier Tasten. 0 kann hier bedenkenlos als Dummy (Füllbyte ohne wirkliche Bedeutung) stehen, da der Akku aufgrund von 620 nie den Wert 0 annehmen wird.

Beinhaltet X nach der Division durch 4 den Wert 2, wird das Inversfeld nach unten bewegt, ist X=3, dann nach oben. Dies können Sie sich an Bild 6 veranschaulichen.

An den Zeilen 740 – 870 sehen wir nun die Verwendung einer Sprungtabelle. Unsere Sprungtabelle ist »SP1LO/SP1HI«. »SP1LO« beinhaltet die Low-, »SP1HI« die High-Bytes der anzuspringenden Routinen. In den Vektor »SPRUNG« wird einfach die Zieladresse geschrieben (740 – 770).

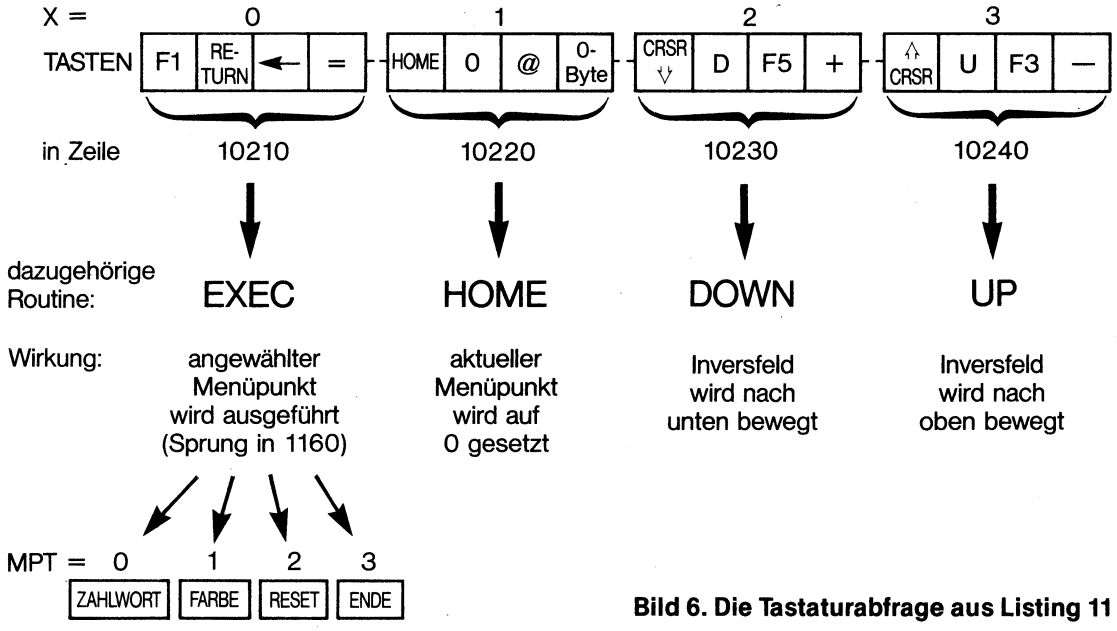

Bild 6. Die Tastaturabfrage aus Listing 11

Die Zuweisungszeile 790 errechnet die Rücksprungadresse des aufzurufenden Unterprogramms. Bei einem RTS soll nämlich zur »HSCHLEIFE« gesprungen werden.

Diese Rücksprungadresse »RUECKSPRUNG« wird auf den Stapel gelegt (830 - 860), zuletzt erfolgt der indirekte Sprung (870). Die über die soeben beschriebene Simulation von JSR (ind) angesprungenen Routinen finden Sie ab Zeile 900. Es wird einfach der aktuelle Menüpunkt »MPT« entsprechend dem Tastendruck geändert, dann wird zur »HSCHLEIFE« gesprungen, die auch die Tabelle »RVSTAB« entsprechend anpaßt.

»EXEC« (1090) holt die Rücksprungadresse vom Stapel (1090 - 1100), da diese Routine nicht als Unterprogramm behandelt werden soll.

Die Zeile 1110 holt den angeforderten Menüpunkt ins X-Register. Dann wird aus »SP2LO/SP2HI« die Adresse der zum Menüpunkt gehörenden Routine geholt und diese über einen gewöhnlichen indirekten Sprung aufgerufen (1160).

Als Routine zu »2« wird einfach die RESET-Routine des Betriebssystems angesprungen, für »3« eignet sich jeder RTS-Befehl, also auch der bei »ENDE« (920).

»ZAHLWORT«, die Routine zu 0, holt eine Zahl als ASCII-Code (1230) und wandelt sie in einen numerischen Wert um (1240 - 1250), indem der ASCII-Code von 0 abgezogen wird. Das Ergebnis landet im X-Register (1260). Ob auch eine Zahl eingegeben wurde, prüfen die Zeilen 1310 - 1330. Bei »ZAHLWORT« (1350) wird das Resultat der Subtraktion in »XSAVE« gesichert, der Text »IN WORTEN« ausgegeben und das X-Register wieder geholt.

Die Tabelle »ZWLO/ZWHI« enthält die Adressen, ab denen die Texte der Zahlwörter als ASCII-Code stehen. Aus »ZWLO/ZWHI« wird dann diese Adresse geholt (1400 - 1410) und der dort stehende Text ausgegeben (1420). Danach erwartet das Programm noch einen Tastendruck (1440-1450), bevor ins Hauptmenü verzweigt wird (1460).

Als letzte Routine wird »FARBE« besprochen (1500-1850): Hierzu ist jedoch aufgrund der Ähnlichkeit zu Listing 10 nicht viel zu erläutern. Bei 1820 steht im X-Register der Code der eingegebenen Farbe (= Position der Prüfsumme innerhalb der Tabelle »PRUEFSUMMEN«). Dieser muß nur noch in die entsprechenden VIC-Register geschrieben werden (1820-1830). Ab Zeile 10000 stehen dann die Tabellen. Wenn Sie die Tabellen angesehen haben, sollten Sie durchaus noch einmal den Quelltext bis 10 000 betrachten und die hier endende Beschreibung des Programms lesen. Denn wenn Sie das Programm »TABELLEN-BEISPIEL« ganz verstanden haben, sind Sie einen großen Schritt in der Assemblerprogrammierung weitergekommen!

Ich könnte mir übrigens vorstellen, daß Sie in Ihren eigenen Programmen jetzt auch eine Menüsteuerung wie die in »TABELLEN-BEISPIEL« einbauen; wie das geht, können Sie dem Programm »TABELLEN-BEISPIEL« entnehmen.

Eine Anmerkung ist wichtig: »TABELLEN-BEISPIEL« kann noch weiter verbessert werden. Sie werden sehen, daß viele Stellen noch optimiert werden können. Insbesondere der Speicherplatzbedarf kann verringert werden.

f) Weitere Anregungen zur Anwendung von Tabellen

Auch die bisherigen Erläuterungen und das Beispielprogramm können die Kreativität des Programmierers nicht ersetzen, sondern nur die Programmierung erleichtern. Aus diesem Grund möchte ich Ihnen noch einige Beispiele nennen, wie sich Tabellen sinnvoll verwerten lassen.

- Ein Anwenderprogramm, das aus Menüs und Untermenüs besteht, sollte in einer Tabelle die Adressen der Menüs/Untermenüs speichern.
- Spiele müssen oft viele Spritebewegungen, die immer gleich sind, durchführen. Es empfiehlt sich, die Spritebewegungen als Koordinaten in einer Tabelle abzulegen.
- Bei Software-Interfaces müssen viele Umrechnungen erfolgen. Durch eine Umwandlungstabelle können diese stark beschleunigt werden.
- Naturwissenschaftlich orientierte Programme müssen verschiedene Maße umrechnen. Die Umrechnungswerte können in einer Tabelle untergebracht werden.

Dies soll nur eine Anregung sein. Ich wüßte aber kein komplexes Programm, das sich nicht durch den gezielten Einsatz von Tabellen vereinfachen und beschleunigen ließe.

Texteinschub #1: Fließkommazahlen

Im Text wurde ein Verfahren vorgestellt, um eine Zahl ins MFLPT-Format (MELPT=Memory floating point) umzuwandeln. Das 5 Byte lange Ergebnis dieser Umwandlung kann man dann als KONSTANTE handhaben. Konstanten sind feste, vorausberechnete Werte, die man mit Hilfe der Routine »MEMFAC« in den FAC (Flieskomma-AKKU) kopieren kann. Für viele Werte ist es jedoch überflüssig, die Umwandlung durchzuführen und eine entsprechende Tabelle anzulegen, da sie schon im ROM vorhanden sind. Im Kurs »Assembler ist keine Alchimie« wurden solche Konstanten mitsamt ihrer Adressen schon in einer Tabelle vorgestellt.

Um mit Konstanten (für die Rechenroutinen macht es keinen Unterschied, ob diese im RAM oder im ROM stehen) zu rechnen, kann man diese wie gesagt, in den FAC kopieren und alle weiteren Operationen auf diesen beziehen. Dies war in Listing 8 bei den Funktionen SQR und LOGNAT ausreichend.

Oft möchte man aber den Inhalt des FAC nicht mit einer Funktion wie SQR behandeln, sondern mit anderen Konstanten addieren, multiplizieren und so weiter.

Dafür möchte ich Ihnen im folgenden weitere Interpreter-Routinen vorstellen, die das Rechnen mit Konstanten ermöglichen. Da fast immer in den Akku das Low-, ins Y-Register das High-Byte der Adresse, ab der die Konstante abgelegt ist, geladen werden muß, definieren wir noch vorher folgende Makro:

```
-.MA LDAY (ADRESSE)
-          LDA # <(ADRESSE)
-          LDY # >(ADRESSE)
-.RT
```

Nun zu den Routinen, bei deren Parameterübergabe wir uns auf das Makro LDAY stützen wollen:

ADDMEM	FAC+Konstante → FAC	...LDAY (KONSTANTE) JSR $B867
ADD0,5	FAC+0.5 → FAC	JSR $B849
SUBMEM	Konstante-FAC → FAC	...LDAY (KONSTANTE) JSR $B850
MULMEM	Konstante*FAC → FAC	...LDAY (KONSTANTE) JSR $BA28
MULT10	FAC*10 → FAC	JSR $BAE2
DIVMEM	Konstante/FAC → FAC	...LDAY (KONSTANTE) JSR $BB0F
DIVS10	FAC/10 → FAC	JSR $BAFE
CMPMEM	vergleicht Konstante mit FAC FAC<Konstante: Akku=$FF FAC=Konstante: Akku=$00 FAC>Konstante: Akku=$01	...LDAY (KONSTANTE) JSR $BC5B
POTMEM	Konstante ↑FAC → FAC	...LDAY (KONSTANTE) JSR $BF78
POTE	e↑FAC → FAC	JSR $BFED
MEMFAC	holt Konstante in FAC	...LDAY (KONSTANTE) JSR $BBA2
FACMEM	FAC ab Konstante als MFLPT-Zahl ablegen	LDX # <(KONSTANTE) LDY # >(KONSTANTE) JSR $BBD7
FACOUT	gibt FAC aus	JSR $AABC

7. Die Initialisierung

»Initialisierung« nennt man eine Routine, die vor einem Programmteil (meist einer Schleife) steht und diese vorbereitet. Die Initialisierung wird nur einmal, eine Schleife aber mehrfach durchlaufen. Deshalb bringt es einen Geschwindigkeitszuwachs, wenn die Initialisierung der Schleife Arbeit abnimmt.

Ein Beispiel: Wenn ein Basic-Programm mit »RUN« gestartet wird, werden alle Variablen gelöscht, Files geschlossen und die Adressen, ab denen die Variablen abgelegt werden dürfen, errechnet. Dies ist die Initialisierung der Interpreterschleife. Dann wird Byte für Byte des Basic-Programms eingelesen und bearbeitet.

Muß im gerade übersetzten Befehl ein Sprung (GOTO 500 oder ähnliches) durchgeführt werden, kostet dies bekanntlich viel Zeit, wenn das Sprungziel am Ende eines langen Programms steht. Dies ist darauf zurückzuführen, daß der Interpreter, beginnend mit der ersten Zeile, das ganze Programm nach der Sprungzeile durchsucht, bis er sie gefunden hat.

Diese Berechnung der Adressen wird bei jedem »GOTO« oder »GOSUB« neu durchgeführt.

Viel besser und schneller wäre folgende Vorgehensweise: Bei »RUN« wird zunächst eine Tabelle angelegt, in der die Adressen aller Zeilen enthalten sind. Diese Tabelle könnte zum Beispiel als Array definiert werden. Folgt nun ein Sprung, kann aus der Tabelle die Adresse der Zeile im Speicher geholt werden.

Damit haben wir noch ein wesentliches Merkmal der Initialisierungsroutinen gefunden: Die Initialisierung kann Tabellen anlegen, die dann von der Hauptschleife verarbeitet werden.

Aber nicht nur Tabellen können generiert werden, auch die Berechnung von Flags ist sinnvoll. So merkt sich die »LOAD/VERIFY«-Routine ($FFD5), ob ein Verifizieren oder Laden gewünscht wird. Die Ladeschleife liest dann ein Zeichen von der Floppy oder der Datasette ein und entscheidet erst anschließend, ob das Byte im Speicher abgelegt oder mit dem Speicher verglichen werden soll.

Halten wir also fest, daß Initialisierungsroutinen Schleifen entlasten können. Näher werden wir uns damit beim Thema »Schleifen« beschäftigen.

8. Die Nutzung der Zeropage

In jedem Assembler-Lehrbuch werden die Vorteile der Zeropage-Adressierung gepriesen. Speicherplatzersparnis und hohe Verarbeitungsgeschwindigkeit sind nicht die einzigen Vorzüge; die indirekt-indizierte Adressierung kann nur auf Zeropage-Adressen zugreifen, nicht auf absolute 16-Bit-Adressen. Damit wird der Leser aber schon alleine gelassen. Er erfährt nicht, welche Adressen in der Zeropage für die Praxis geeignet sind. Das wird nun nachgeholt.

Fast die ganze Zeropage wird durch Basic-Interpreter und Betriebssystem belegt. Deshalb führen bestimmte Werte in Zeropage-Adressen oft zum Absturz oder sonstigem Fehlverhalten des Computers.

Wie dies im einzelnen aussieht, erfahren Sie in der Serie »Memory Map mit Wandervorschlägen«, die im 64'er Stammheft erscheint. Nicht nur in Zweifelsfällen stellt diese Serie das optimale Nachschlagewerk dar.

Ich möchte Ihnen nun zeigen, welche Adressen Sie als (Zwischen-)Speicher ohne Schwierigkeiten verwenden können, beziehungsweise was Sie bei Verwendung von Zeropage-Adressen beachten müssen.

a) Adressen, die problemlos verwendet werden können

Auf die Adressen $02 und $FB - $FE wird weder vom

```
,6000  A2 00      LDX #00
,6002  B5 02      LDA   02,X
,6004  9D 00 6F   STA 6F00,X
,6007  E8         INX
,6008  E0 FE      CPX #FE
,600A  D0 F6      BNE 6002
```

Listing 12

```
,6000  A2 FE      LDX #FE
,6002  B5 01      LDA #01
,6004  9D FF 6E   STA 6EFF,X
,6007  CA         DEX
,6008  D0 F8      BNE 6002
```

Listing 13

```
,6000  A2 FE      LDX #FE
,6002  BD FF 6E   LDA 6EFF,X
,6005  95 01      STA #01
,6007  CA         DEX
,6008  D0 F8      BNE 6002
```

Listing 14

```
,6000  A2 34      LDX #34
,6002  B5 16      LDA   16,X
,6004  9D 00 6F   STA 6F00,X
,6007  CA         DEX
,6008  10 F8      BPL 6002
```

Listing 15

```
,6000  A2 34      LDX #34
,6002  BD 00 6F   LDA 6F00,X
,6005  95 16      STA   16,X
,6007  CA         DEX
,6008  10 F8      BPL 6002
```

Listing 16

```
,6000  A2 FF      LDX #FF
,6002  BD 00 01   LDA 0100,X
,6005  9D 00 6F   STA 6F00,X
,6008  CA         DEX
,6009  D0 F7      BNE 6002
,600B  AD 00 01   LDA 0100
,600E  8D 00 6F   STA 6F00
,6011  BA         TSX
,6012  8E 00 70   STX 7000
```

Listing 17

```
,6000  A2 FF      LDX #FF
,6002  BD 00 6F   LDA 6F00,X
,6005  9D 00 01   STA 0100,X
,6008  CA         DEX
,6009  D0 F7      BNE 6002
,600B  AD 00 6F   LDA 6F00
,600E  8D 00 01   STA 0100
,6011  AE 00 70   LDX 7000
,6014  9A         TXS
```

Listing 18

Basic-Interpreter noch vom Betriebssystem zugegriffen. Lediglich bei Initialisierung der Arbeitsspeicher (RESET) werden Sie auf 0 gesetzt.

Für die Praxis heißt das, daß Ihnen die genannten Adressen völlig zur Verfügung stehen.

b) Adressen, die in keiner Weise verwendet werden sollten

Von anderen Adressen hingegen müssen wir unsere Finger lassen. Diese haben entweder elementare Funktionen für Betriebssystem oder CPU, oder werden von beiden dauernd geändert, so daß die Datensicherheit in Frage gestellt ist. Genauer soll hier nicht unterschieden werden.

Belassen Sie die Adressen $00 und $01 unverändert, da sie (siehe Memory Map) für die CPU wichtige Informationen beinhalten und außerdem einige Bits nur durch externe Vorgänge geändert werden.

Das Betriebssystem und der Basic-Interpreter beanspruchen alle bislang ungenannten Adressen.

Von Bildschirmeditor und Tastaturabfrage werden die Adressen $C6 - $F6 beeinflußt. Die Adressen $90 - $C2 dienen der Ein-/Ausgabe-Steuerung mit Peripheriegeräten und der Verwaltung offener Files. Einzige Ausnahme: $A0 - $A2 (interne Uhr). Wenn ein Maschinenprogrammm in ein Basic-Programm eingebaut ist, sind die Adressen $03 - $56 sowie $73 - $8B tabu.

c) Bedingt einsetzbar

Der Vektor $C3/$C4 wird durch RUN/STOP-Restore, RESET oder LOAD beeinflußt. Ansonsten kann mit $C3/$C4 frei verfahren werden.

Ganz Vorsichtige können diesen Vektor auf seinen Ausgangswert $FD30 setzen, sobald das Programm die Adressen $C3/$C4 nicht mehr für eigene Zwecke benötigt.

d) Adressen, die unter Verzicht auf Kassettenbetrieb verwendet werden können

Die folgenden Adressen können verwendet werden, wenn nicht auf RS232 oder Datasette zugegriffen wird.

$9E/$9F, $A5-$A7, $A9-$AB, $B0-$B6, $F7-$FA

Bei anderen Adressen, die sich auf den RS232- oder Kassettenbetrieb beziehen, ist Vorsicht angebracht.

e) Geeignete Zwischenspeicher

Die Adressen $22-$2A und $57-$60 sind sogenannte »verschieden genutzte Arbeitsbereiche«. Sie werden vom Basic-Interpreter vor allem bei arithmetischen Operationen als Zwischenspeicher verwendet. Als solche Zwischenspeicher können wir sie auch verwenden. Sobald allerdings bestimmte Interpreterroutinen aufgerufen werden, können die Inhalte dieser Adressen verlorengehen. Eine längerfristige Aufbewahrung von Daten in diesen Adressen ist zwar nicht möglich, andererseits können wir aber durch Schreibzugriffe auf diese Adressen das Betriebssystem oder den Basic-Interpreter nicht stören.

Zu sagen wäre noch, daß die Adressen $57 - $60 den wichtigen Routinen BLTUC und UMULT (siehe »Assembler ist keine Alchimie«) als Zwischenspeicher dienen.

f) Zeropage kopieren

Zum Abschluß dieses Abschnittes über die Nutzung der Zeropage möchte ich Ihnen noch einen kleinen Trick verraten, der von einigen professionellen Programmen angewandt wird.

Wir sichern die Zeropage-Inhalte in einem anderen Bereich, zum Beispiel von $6F00 an.

Dann können wir viele Adressen in der Zeropage nutzen, sofern wir keine Interpreter- oder Betriebssystemroutine aufrufen. Danach schreiben wir die Zeropage wieder von der Kopie, zum Beispiel von $6F00, zurück und können wie gewöhnlich fortfahren.

Die Adressen 0 und 1 kopieren wir nicht, weil diese nach wie vor für solche Zwecke nutzlos sind. Ebenso könnten wir

```
,6000  A9 D2     LDA #D2
,6002  85 14     STA   14
,6004  A9 3F     LDA #3F
,6006  85 15     STA   15
,6008  A0 00     LDY #00
,600A  B1 14     LDA  (14),Y
,600C  49 FF     EOR #FF
,600E  91 14     STA  (14),Y
,6010  E6 14     INC   14
,6012  D0 02     BNE 6016
,6014  E6 15     INC   15
,6016  A5 14     LDA   14
,6018  C9 60     CMP #60
,601A  A5 15     LDA   15
,601C  E9 47     SBC #47
,601E  90 EA     BCC 600A
```

Listing 19

```
,6000  A9 5F     LDA #5F
,6002  85 14     STA   14
,6004  A9 47     LDA #47
,6006  85 15     STA   15
,6008  A0 00     LDY #00
,600A  B1 14     LDA  (14),Y
,600C  49 FF     EOR #FF
,600E  91 14     STA  (14),Y
,6010  A5 14     LDA   14
,6012  D0 02     BNE 6016
,6014  C6 15     DEC   15
,6016  C6 14     DEC   14
,6018  A5 14     LDA   14
,601A  C9 D2     CMP #D2
,601C  A5 15     LDA   15
,601E  E9 3F     SBC #3F
,6020  B0 E8     BCS 600A
```

Listing 20

```
,6000  A9 00     LDA #00
,6002  85 14     STA   14
,6004  A9 20     LDA #20
,6006  85 15     STA   15
,6008  A0 00     LDY #00
,600A  B1 14     LDA  (14),Y
,600C  49 FF     EOR #FF
,600E  91 14     STA  (14),Y
,6010  C8        INY
,6011  D0 F7     BNE 600A
,6013  E6 15     INC   15
,6015  A5 15     LDA   15
,6017  C9 40     CMP #40
,6019  D0 EF     BNE 600A
```

Listing 21

```
,6000  A9 00     LDA #00
,6002  85 14     STA   14
,6004  A8        TAY
,6005  A9 20     LDA #20
,6007  85 15     STA   15
,6009  AA        TAX
,600A  B1 14     LDA  (14),Y
,600C  49 FF     EOR #FF
,600E  91 14     STA  (14),Y
,6010  C8        INY
,6011  D0 F7     BNE 600A
,6013  E6 15     INC   15
,6015  CA        DEX
,6016  D0 F2     BNE 600A
```

Listing 22

nur einzelne Bereiche kopieren (zum Beispiel die Zeiger für Basic-Programme $16 - $4A). Dann dürfen wir aber auch nur diesen Bereich verändern.

Wenn wir nun den Bereich $02 - $FF kopieren, stehen uns folgende Adressen zur Verfügung:

$03-$06, $14-$86, $71-$8A, $C3/$C4, $FB-$FF

Diese Adressen können Sie nur so lange verwenden, bis eine Routine des Betriebssystems oder Basic-Interpreters aufgerufen wird. Davor muß die alte Zeropage zurückgeschrieben werden.

Da Sie auf diese Weise viel Speicherplatz in der Zeropage gewonnen haben, ist es sogar möglich, eine Tabelle aus Geschwindigkeitsgründen in die Zeropage zu verlegen. Damit steigt auch der Wert der indiziert-indirekten Adressierung erheblich.

Dennoch ist der Speicherplatz in der Zeropage begrenzt. Überlegen Sie sich also, auf welche Werte besonders schnell zugegriffen werden muß und schreiben Sie vorzugsweise diese in die Zeropage.

```
70   -.BA $C000
80   -.LI 1,3,0
90   -;
100  -; ********************************
110  -; *    QUELLTEXTE (HYPRA-ASS)    *
120  -; *    ======================    *
130  -; *                              *
140  -; * FUER VERSCHIEDENE SCHLEIFEN *
150  -; *                              *
160  -; * 28.08.85 BY FLORIAN MUELLER *
170  -; *                              *
180  -; ********************************
190  -;
200  -;
210  -; QUELLTEXT ZU LISTING 1
220  -; ======================
230  -;
240  -.EQ ANFANGSADRESSE = $02
250  -.EQ ENDADRESSE = $FF
260  -.EQ ZIELBEREICH = $6F00
270  -;
280  -           LDX #0
290  -SCHLEIFE1 LDA ANFANGSADRESSE,X
300  -           STA ZIELBEREICH,X
310  -           INX
320  -           CPX #(ENDADRESSE+1-ANFANGSADRESSE)
330  -           BNE SCHLEIFE1
340  -;
350  -;
360  -; QUELLTEXT ZU LISTING 2
370  -; ======================
380  -;
390  -.EQ ANFANGSADRESSE = $02
400  -.EQ ENDADRESSE = $FF
410  -.EQ ZIELBEREICH = $6F00
420  -;
430  -           LDX #(ENDADRESSE+1-ANFANGSADRESSE)
440  -SCHLEIFE2 LDA ANFANGSADRESSE-1,X
450  -           STA ZIELBEREICH-1,X
460  -           DEX              ; DEKREMENTIERBEFEHL
470  -           BNE SCHLEIFE2
480  -;
490  -;
500  -; QUELLTEXT ZU LISTING 4
510  -; ======================
520  -;
530  -.EQ ANFANGSADRESSE = $16
540  -.EQ ENDADRESSE = $4A
550  -.EQ ZIELBEREICH = $6F00
560  -;
570  -           LDX #(ENDADRESSE-ANFANGSADRESSE)
580  -SCHLEIFE3 LDA ANFANGSADRESSE,X
590  -           STA ZIELBEREICH,X
600  -           DEX
610  -           BPL SCHLEIFE3  ; PRUEFT N-FLAG
620  -;
630  -;
640  -; QUELLTEXT ZU LISTING 8
650  -; ======================
660  -;
670  -.EQ ANFANGSADRESSE = $3FD2
680  -.EQ ENDADRESSE = $475F
690  -.EQ ZAEHLER = $14
700  -;
```

Listing 23

```
710  -           LDA #<(ANFANGSADRESSE)
720  -           STA ZAEHLER
730  -           LDA #>(ANFANGSADRESSE)
740  -           STA ZAEHLER+1
750  -           LDY #0
760  -SCHLEIFE4 LDA (ZAEHLER),Y
770  -           EOR #$FF
780  -           STA (ZAEHLER),Y
790  -           INC ZAEHLER
800  -           BNE WEITER
810  -           INC ZAEHLER+1
820  -WEITER    LDA ZAEHLER
830  -           CMP #<(ENDADRESSE+1)
840  -           LDA ZAEHLER+1
850  -           SBC #>(ENDADRESSE+1)
860  -           BCC SCHLEIFE4
870  -;
880  -;
890  -; QUELLTEXT ZU LISTING 10
900  -; ======================
910  -;
920  -.EQ ANFANGSADRESSE = $2000
930  -.EQ ENDADRESSE  = $3FFF
940  -.EQ ZAEHLER = $14
950  -;
960  -           LDA #<(ANFANGSADRESSE)
970  -           STA ZAEHLER
980  -           LDA #>(ANFANGSADRESSE)
990  -           STA ZAEHLER+1
1000 -           LDY #0
1010 -SCHLEIFE5 LDA (ZAEHLER),Y
1020 -           EOR #$FF
1030 -           STA (ZAEHLER),Y
1040 -           INY
1050 -           BNE SCHLEIFE5
1060 -           INC ZAEHLER+1
1070 -           LDA ZAEHLER+1
1080 -           CMP #>(ENDADRESSE+1)
1090 -           BNE SCHLEIFE5
1100 -;
1110 -;
1120 -; QUELLTEXT ZU EINER SCHLEIFE,
1130 -; DIE DEN BEREICH $3FD2-$47D1
1140 -; KOMPLEMENTIERT
1150 -;
1160 -.EQ ANFANGSADRESSE = $3FD2
1170 -.EQ ENDADRESSE = $47D1
1180 -.EQ ZAEHLER = $14
1190 -;
1200 -           LDA #<(ANFANGSADRESSE)
1210 -           STA ZAEHLER
1220 -           LDA #>(ANFANGSADRESSE)
1230 -           STA ZAEHLER+1
1240 -           LDX #>(ENDADRESSE+1-ANFANGSADRESSE)
1250 -           LDY #0
1260 -SCHLEIFE6 LDA (ZAEHLER),Y
1270 -           EOR #$FF
1280 -           STA (ZAEHLER),Y
1290 -           INY
1300 -           BNE SCHLEIFE6
1310 -           INC ZAEHLER+1
1320 -           DEX
1330 -           BNE SCHLEIFE6
1340 -;
1350 -; ENDE VON LISTING 12
```

9. Schleifenprogrammierung

Zunächst befassen wir uns mit Schleifen, die maximal 256mal durchlaufen werden.

Typ a: Schleifen mit maximal 256 Durchläufen

Da 256 verschiedene Zahlen mit einem 8-Bit-Prozessor dargestellt werden können, verwendet man hier das X- (oder Y-) Register als Schleifenzähler. In Listing 12 sehen Sie die einfachste Form einer Schleife, die die Zeropage-Adressen $02 - $FF nach $6F00 kopiert.

Da der Schleifenzähler X in Listing 12 INKREMENTIERT wird, haben wir es mit einer INKREMENTIERSCHLEIFE zu tun. Nach dem Inkrementieren (»6007 INX«) wird durch »6008 CPX # FE« überprüft, ob die Schleife beendet werden kann. Eine eingehendere Beschreibung des Programmablaufs erübrigt sich.

Für Schleifen des Typs a (maximal 256 Durchläufe) ist es aber meist vorteilhaft, eine DEKREMENTIERSCHLEIFE zu verwenden. Wie eine solche Schleife programmiert wird, sehen wir an Listing 13.

Listing 13 unterscheidet sich in der Wirkung nicht von Listing 12, obwohl man dies nicht unbedingt auf den ersten Blick erkennt. Deshalb soll dieses Listing näher besprochen werden. In Zeile 6000 erhält das X-Register den Inhalt $FE. Durch »6002 LDA 01,X« wird damit das letzte Byte der Zeropage, nämlich $FF, zuerst gelesen und nach $70FE geschrieben. Dann wird X dekrementiert. Ist X noch nicht 0, so wird die Schleife erneut durchlaufen.

Der niedrigste X-Wert innerhalb der Schleife ist folglich 1; aufgrund von »6002 LDA 01,X« ist $02 die niedrigste Zeropage-Adresse, die kopiert wird. In Listing 12 ist 0 der niedrigste X-Wert. Die niedrigste Adresse aufgrund von »6002 LDA 02,X« ist also auch $02 (stimmt auffällig). Warum $FF die höchste kopierte Zeropage-Adresse ist, können Sie nun selbst den Listings 12 und 13 entnehmen.

Listing 14 ist eine Dekrementierschleife, die die Kopie der Zeropage wieder von $6F00 nach $02 zurückholt.

Der Vorteil von Dekrementierschleifen beim Typ a ist, daß zum Erkennen der Abbruchbedingung (X=0) kein Vergleichsbefehl erforderlich ist, weil nach dem DEX-Befehl automatisch das Z-Flag gesetzt wird, wenn X Null wird.

Das Entfallen des Vergleichsbefehls »CPX #« bringt eine Ersparnis von 2 Byte Speicherplatz sowie insgesamt 508 Taktzyklen Rechenzeit. Da jedoch bei 6004 eine Seitenüberschreitung (eine Seite entspricht 256 Byte) vorliegt, schrumpft der Zeitgewinn auf 254 Taktzyklen (dies ließe sich aber vermeiden, indem wir die Zeropage nach $6F01 kopieren, womit durch »6004 STA $6F00,X« keine Seitenüberschreitung auftreten würde).

Nun wollen wir noch einen Sonderfall behandeln:

Dekrementierschleifen vom Typ a, bei denen der Ausgangswert für X < 129 ist.

In Listing 15 sehen Sie eine Schleife, die den Bereich $16 - $4A nach $6F00 kopiert, Listing 16 schreibt die Werte von $6F00 zurück nach $16. Selbstverständlich hätten wir das Problem auch so lösen können wie in Listing 13. Wir wollen aber noch eine andere Konstruktion von Dekrementierschleifen kennenlernen, die in diesem Sonderfall möglich ist. Besprechen wir also Listing 15.

Bei 6000 wird ins X-Register die Zahl geladen, die man zu $16 addieren muß, um $4A zu erhalten. Dadurch wird zunächst bei 6002 die Adresse $4A gelesen und nach $6F34 geschrieben. Bei 6007 wird dekrementiert. Neu ist der Verzweigungsbefehl: es wird das N-Flag überprüft. Ist X = $FF, wird das N-Flag gesetzt und »6008 BPL 6002« beendet die Schleife. Der niedrigste Wert von X, der innerhalb der Schleife vorkommt, ist demnach $00.

Der BPL-Befehl funktioniert nur, wenn der Ausgangswert

```
,6000  A0 00     LDY #00
,6002  B9 00 20  LDA 2000,Y
,6005  49 FF     EOR #FF
,6007  99 00 20  STA 2000,Y
,600A  C8        INY
,600B  D0 F5     BNE 6002
,600D  EE 04 60  INC 6004
,6010  EE 09 60  INC 6009
,6013  AD 09 60  LDA 6009
,6016  C9 40     CMP #40
,6018  D0 E8     BNE 6002
```

Listing 24

```
,6000  A0 00     LDY #00
,6002  B9 00 40  LDA 4000,Y
,6005  49 FF     EOR #FF
,6007  99 00 40  STA 4000,Y
,600A  C8        INY
,600B  D0 F5     BNE 6002
,600D  EE 04 60  INC 6004
,6010  EE 09 60  INC 6009
,6013  AD 09 60  LDA 6009
,6016  C9 40     CMP #40
,6018  D0 E8     BNE 6002
```

Listing 25

```
,6000  A9 00     LDA #00
,6002  8D 13 60  STA 6013
,6005  8D 18 60  STA 6018
,6008  A9 20     LDA #20
,600A  8D 14 60  STA 6014
,600D  8D 19 60  STA 6019
,6010  A0 00     LDY #00
,6012  B9 FF FF  LDA FFFF,Y
,6015  49 FF     EOR #FF
,6017  99 FF FF  STA FFFF,Y
,601A  C8        INY
,601B  D0 F5     BNE 6012
,601D  EE 14 60  INC 6014
,6020  EE 19 60  INC 6019
,6023  AD 19 60  LDA 6019
,6026  C9 40     CMP #40
,6028  D0 E8     BNE 6012
```

Listing 26

von X <129 ist. Andernfalls wäre nämlich nach dem Dekrementieren X>127 und damit das N-Flag gesetzt. Dies aber hätte zur Folge, daß die Schleife nur 1mal durchlaufen würde.

Zur soeben behandelten Schleifenkonstruktion sind noch zwei Dinge zu sagen; erstens, daß sie nur in diesem Sonderfall (X<129) möglich ist, und zweitens, daß sie nicht effektiver als eine Lösung wie in Listing 13 ist.

Allgemeine Gültigkeit hat aber folgende Regel für Schleifen vom Typ a:

> Bei Schleifen vom Typ a ist Dekrementieren effektiver als Inkrementieren, solange die Durchlaufzahl nicht 255 überschreitet.
> Bei 256 Durchläufen erweist sich Inkrementieren oft als besser.

An Listing 17 sehen wir ein Beispiel für den letzten Satz der Regel. Listing 17 kopiert die letzten 256 Speicherplätze des Stapels ($0100 - $01FF) und den Stapelzeiger nach $6F00 - $7000. Listing 18 schreibt den Stapel wieder zurück.

Die Dekrementierschleife (6000 - 600A) kopiert nun den Bereich $0101 - $01FF, $0100 wird nicht übertragen. Dies geschieht in 600B - 600F. Eine andere Möglichkeit wäre ein zeitraubender CPX #FF-Befehl nach »6008 DEX«.

6011 - 6013 sichert schließlich noch das SP-Register.

Hier ist in der Tat eine Inkrementierschleife besser. Ändern wir Listing 17 also in Listing 17a:

```
        LDX #00
- LOOP  LDA 0100,X
-       STA 6F00,X
        INX          ;(!!)
-       BNE LOOP
-       TSX
-       STX 7000
```

Analog ergibt sich Listing 18a:

```
        LDX #00
- LOOP  LDA 6F00,X
-       STA 0100,X
        INX          ;(!!)
-       BNE LOOP
-       LDX 7000
-       TXS
```

In den Listings 17a und 18a habe ich diejenigen Befehle, die sich in der symbolischen Darstellung nicht von den Listings 17 und 18 unterscheiden, mit einem »-« markiert.

Typ b: Schleifen mit mehr als 256 Durchläufen

Während Schleifen des Typs a meist so schnell abgearbeitet werden, daß man es gar nicht bemerkt, dauern Typ-b-Schleifen oft eine oder mehrere Sekunden.

Deswegen wollen wir hier versuchen, den Zeitbedarf von Typ-b-Schleifen zu verringern.

Unsere erste Typ-b-Schleife (Listing 19) soll den Bereich von $3FD2 bis $475F invertieren (= EOR #FF-verknüpfen, aus jeder 1 wird eine 0 und umgekehrt). Da hierfür ein 8-Bit-Indexregister nicht ausreicht, benötigen wir einen 16-Bit-Zähler, nämlich $14/$15. Dieser soll immer die Adresse beinhalten, die invertiert wird. In diesen Zähler schreibt die Initialisierung der Schleife den Startwert $3FD2 (siehe $6000 - $6007).

Da es beim 6510 keine indirekte Adressierung für LDA/STA gibt, sondern nur die indirekt-indizierte oder indiziert-indirekte, müssen wir auf eine dieser Adressierungen ausweichen und den Index auf 0 setzen (»6008 LDY #00«).

Bei $600A beginnt die Schleife: der Wert wird eingelesen, mit $FF EOR-verknüpft und zurückgeschrieben. Nun wird der 16-Bit-Zähler $14/$15 erhöht (6010 - 6015). Dann wird

```
,6000  A9 D2     LDA #D2
,6002  8D 11 60  STA 6011
,6005  8D 16 60  STA 6016
,6008  A9 3F     LDA #3F
,600A  8D 12 60  STA 6012
,600D  8D 17 60  STA 6017
,6010  AD 00 00  LDA 0000
,6013  49 FF     EOR #FF
,6015  8D 00 00  STA 0000
,6018  EE 11 60  INC 6011
,601B  EE 16 60  INC 6016
,601E  D0 06     BNE 6026
,6020  EE 12 60  INC 6012
,6023  EE 17 60  INC 6017
,6026  AD 11 60  LDA 6011
,6029  C9 60     CMP #60
,602B  AD 12 60  LDA 6012
,602E  E9 47     SBC #47
,6030  90 DE     BCC 6010
```

Listing 27

```
,6000  A2 00     LDX #00
,6002  8E 11 60  STX 6011
,6005  8E 14 60  STX 6014
,6008  A2 A0     LDX #A0
,600A  8E 12 60  STX 6012
,600D  8E 15 60  STX 6015
,6010  AE 00 00  LDX 0000
,6013  8E 00 00  STX 0000
,6016  EE 11 60  INC 6011
,6019  EE 14 60  INC 6014
,601C  D0 F2     BNE 6010
,601E  EE 12 60  INC 6012
,6021  EE 15 60  INC 6015
,6024  AE 12 60  LDX 6012
,6027  E0 C0     CPX #C0
,6029  D0 E5     BNE 6010
```

Listing 28

```
80   -.BA $6000
90   -.LI 1,3,0
100  -;
110  -; HYPRA-ASS-QUELLTEXT ZU EINER
120  -; SELBSTMODIFIZIERENDEN SCHLEIFE
130  -; (ARBEITET WIE LISTING 5)
140  -;
150  -; 1985 BY FLORIAN MUELLER
160  -;
170  -;
180  -.GL START = $A000
190  -.GL ENDE  = $BFFF
200  -;
210  -          LDX #<(START)
220  -          STX MOD1+1
230  -          STX MOD2+1
240  -          LDX #>(START)
250  -          STX MOD1+2
260  -          STX MOD2+2
270  -MOD1      LDX $FFFF
280  -MOD2      STX $FFFF
290  -          INC MOD1+1
300  -          INC MOD2+1
310  -          BNE MOD1
320  -          INC MOD1+2
330  -          INC MOD2+2
340  -          LDX MOD1+2
350  -          CPX #>(ENDE+1)
360  -          BNE MOD1
```

Listing 29

```
100  -.BA $0801
110  -.OB "LOADER-MAKER 64,P,W"
120  -;
130  -;
140  -; ****************************
150  -; *                          *
160  -; *  L O A D E R - M A K E R  *
170  -; *                          *
180  -; ****************************
190  -; *                          *
200  -; *   EIN PROGRAMMGENERATOR   *
210  -; *                          *
220  -; *    VON FLORIAN MUELLER    *
230  -; *                          *
240  -; ****************************
250  -;
260  -;
270  -;
280  -.GL BASIN  = $FFCF
290  -.GL SETPAR = $FFBA
300  -.GL SETNAM = $FFBD
310  -.GL LOAD   = $FFD5
320  -.GL READY  = $A474
330  -.GL NUMOUT = $BDCD
340  -.GL TASTPF = 631 ; TASTATURPUFFER
350  -.GL ANZAHL = 198 ; ENTHAELT ANZAHL
360  -;                  DER ZEICHEN IM
370  -;                  TASTATURPUFFER
380  -.GL KASSPF = 828 ; KASSETTENPUFFER
390  -;
400  -;
410  -.MA PRINT (TEXT)
420  -           LDA #<(TEXT)    ; MAKRO
430  -           LDY #>(TEXT)    ; FUER
440  -           JSR $AB1E       ; TEXTAUSGABE
450  -.RT
460  -;
470  -;
480  -;
490  -;
500  -.WO LINK+1 ; LINKPOINTER
510  -.WO 1985   ; ZEILENNUMMER
520  -.BY $9E    ; TOKEN FUER "SYS"
530  -           .TX "2061"
540  -LINK       .BY 0,0,0       ; ENDMARKIERUNG
550  -;                            DER BASIC-ZEILE
560  -;
570  -SYSTEM     LDX #0          ; FLAG FUER SYSTEM-
580  -           STX $9D         ; MELDUNGEN SETZEN
590  -;
600  -           LDX #$49        ; DEKR.-ZAEHLER
610  -SCHLEIFE1  LDA ABLAGE,X    ; LADEROUTINE
620  -           STA KASSPF,X    ; VON ABLAGE IN
630  -           DEX             ; DEN BEREICH
640  -           BPL SCHLEIFE1   ; KOPIEREN, IN
650  -;                            DEM SIE LAEUFT
660  -           JMP KASSPF      ; & STARTEN
670  -;
680  -;
690  -; ES FOLGT DIE LADEROUTINE, DIE HIER
700  -; AN FALSCHER STELLE ABGELEGT IST UND
710  -; VON DER "SCHLEIFE1" (600-640) IN
720  -; DEN ORIGINALBEREICH GESCHRIEBEN WIRD.
730  -;
740  -ABLAGE     LDA #1          ; FILENUMMER #1
750  -           TAY             ; SEKUNDAERADRESSE #1
760  -GERAETENR  LDX #0          ; GERAETEADRESSE #?
770  -           JSR SETPAR      ; PARAMETER SETZEN
780  -;
790  -LAENGE     LDA #0          ; LAENGE DES FILENAMEN
800  -           LDX #<($35C)    ; ADRESSE DES
810  -           LDY #>($35C)    ; FILENAMEN: $035C
820  -           JSR SETNAM      ; NAMEN SETZEN
830  -;
840  -           LDA #0          ; FLAG FUER "LADEN"
850  -           JSR LOAD
860  -;
870  -FEHLER     BCS LOADERROR   ; LADEFEHLER?
880  -START      JMP 0           ; ZUR STARTADRESSE
890  -LOADERROR  LDX #$1D        ; "LOAD ERROR"
900  -           JMP ($300)      ; AUSGEBEN
910  -;
920  -NAME       .BY 0,0,0,0     ; 16 BYTES
930  -           .BY 0,0,0,0     ; FUER FILENAMEN
940  -           .BY 0,0,0,0     ; RESERVIEREN
950  -           .BY 0,0,0,0
960  -;
970  -BASIC      STX $2D         ; POINTER FUER
980  -           STY $2E         ; PROGRAMMENDE SETZEN
990  -           JSR $E544       ; = PRINT CHR$(147)
1000 -           LDX #3          ; 3 BYTES IN
1010 -           STX ANZAHL      ; TASTATURPUFFER
1020 -;
1030 -SCHLEIFE2  LDA $0383,X     ; AUS DER TABELLE
1040 -           STA TASTPF,X    ; IN ZEILE 1100
1050 -           DEX             ; KOPIEREN
1060 -           BPL SCHLEIFE2
1070 -;
```

Listing 30

```
1080 -           JMP READY       ; WARMSTART
1090 -;
1100 -.BY "R",$D5,13 ; "R",SHIFT U,RETURN
1110 -;
1120 -; HIER ENDET DER PROGRAMMTEIL,
1130 -; DER MODIFIZIERT WIRD.
1140 -; ES FOLGT DIE MODIFIKATIONSROUTINE:
1150 -;
1160 -MDFIKATOR JSR $E544       ; = PRINT CHR$(147)
1170 -...PRINT (TEXT1)
1180 -; STARTADRESSE HOLEN
1190 -;
1200 -           JSR $AEFD       ; PRUEFT AUF KOMMA
1210 -           JSR $AD8A       ; HOLT PARAMETER
1220 -           JSR $B7F7       ; NACH $14/$15
1230 -;
1240 -           LDX $14         ; STARTADRESSE
1250 -           LDA $15         ; HOLEN,
1260 -           STX START+1     ; IM PROGRAMM
1270 -           STA START+2     ; ABLEGEN UND
1280 -           JSR NUMOUT      ; UND AUSGEBEN
1290 -;
1300 -;
1310 -; NUN WIRD NOCH DER ZU MODIFIZIERENDE
1320 -; PROGRAMMTEIL IN DEN AUSGANGSZUSTAND
1330 -; GEBRACHT:
1340 -;
1350 -           LDX #15         ; NAMEN MIT NULL-BYTES
1360 -           LDA #0          ; BELEGEN
1370 -SCHLEIFE3 STA NAME,X       ; DURCH EINE
1380 -           DEX             ; DEKREMENTIER-
1390 -           BPL SCHLEIFE3   ; SCHLEIFE
1400 -;
1410 -           STA SYSTEM+1    ; KEINE SYSTEMMELDUNGEN
1420 -;
1430 -           LDA #3          ; SPRUNGWEITE = 3
1440 -           STA FEHLER+1
1450 -;
1460 -           LDA #$A2        ; OPCODE FUER "LDX #"
1470 -           STA GERAETENR
1480 -;
1490 -;
1500 -; AN DIESER STELLE IST DAS "GERUEST"
1510 -; (DER ZU MODIFIZIERENDE TEIL)
1520 -; IM AUSGANGSZUSTAND
1530 -;
1540 -;
1550 -; EINGABE DES FILENAMEN
1560 -; ======================
1570 -;
1580 -...PRINT (TEXT2)
1590 -           LDX #0          ; ZAEHLER AUF 0
1600 -SCHLEIFE4 JSR BASIN
1610 -           CMP #13         ; ENDE DER EINGABE?
1620 -           BEQ WEITER1     ; JA=>WEITER
1630 -           STA NAME,X      ; BYTE ABLEGEN
1640 -           INX
1650 -           CPX #16         ; 16 ZEICHEN MAX.
1660 -           BNE SCHLEIFE4   ; NAECHSTES ZEICHEN
1670 -;
1680 -; WENN DIESE STELLE DURCHLAUFEN WIRD,
1690 -; HAT DAS X-REGISTER DEN WERT 16.
1700 -;
1710 -; BEI "WEITER1" HINGEGEN KANN ES AUFGRUND
1720 -; DES BRANCH-BEFEHLS "BEQ WEITER1"
1730 -; UNTERSCHIEDLICHE WERTE HABEN.
1740 -;
1750 -WEITER1   STX LAENGE+1
1760 -;
1770 -;
1780 -; EINGABE DER GERAETEADRESSE
1790 -; ==========================
1800 -;
1810 -...PRINT (TEXT3)
1820 -           JSR BASIN       ; HOLT ZEICHEN
1830 -           SEC             ; VOR SUBTRAKTION
1840 -           SBC #"0"        ; IM AKKU STEHT JETZT
1850 -;                            DIE ZAHL
1860 -;
1870 -           STA GERAETENR+1; ABLEGEN
1880 -           BNE WEITER2     ; GERAET<>0 : WEITER
1890 -; DA ALS GERAETENUMMER 0 EINGEGEBEN
1900 -; WURDE, MUSS DER GESAMTE BEFEHL
1910 -; "LDX #GERAET" IN "LDX $BA"
1920 -; UMGEWAENDELT WERDEN, DAMIT DAS
1930 -; NACHLADEN VON DEM GERAET ERFOLGT,
1940 -; VON DEM DER LADER EINGELESEN WIRD.
1950 -;
1960 -           LDA #$A6        ; OPCODE FUER "LDX ZP"
1970 -           STA GERAETENR
1980 -           LDA #$BA        ; "LDX $BA"
1990 -           STA GERAETENR+1; GENERIEREN
2000 -;
2010 -;
2020 -; MASCHINENPROGRAMM (J/N)?
2030 -; ========================
2040 -;
2050 -WEITER2   ... PRINT(TEXT4)
```

Ergänzen Sie jetzt Ihre 64'er-Sammlung

ACHTUNG: Die Ausgaben 5, 6, 7 und 11/84 sind vergriffen und nicht mehr lieferbar!

Schaffen Sie sich ein interessantes Nachschlagewerk und gleichzeitig ein wertvolles Archiv!

Kennen Sie alle Ausgaben von 64'er? Suchen Sie einen ganz bestimmten Testbericht? Oder haben Sie einen Teil eines interessanten Kurses versäumt? Suchen Sie nach einer speziellen Anwendung?

Damit Sie jetzt fehlende Hefte mit »Ihrem« Artikel nachbestellen können, finden Sie auf diesen Seiten eine Zusammenstellung aller wesentlichen Artikel von Ausgabe 4/84 bis Ausgabe 3/85.

Und so kommen Sie schnell an die noch lieferbaren Ausgaben: Prüfen Sie, welche Ausgabe in Ihrer Sammlung noch fehlt, oder welches Thema Sie interessiert. Tragen Sie die Nummer dieser Ausgabe und das Erscheinungsjahr (z.B. 2/85) auf dem Bestellabschnitt der hier eingehefteten Bestell-Zahlkarte ein. Die ausgefüllte Zahlkarte einfach heraustrennen und Rechnungsbetrag beim nächsten Postamt einzahlen. Ihre Bestellung wird nach Zahlungseingang umgehend zur Auslieferung gebracht.

Alle Beiträge sind in der Regel für den C 64, sofern nicht anders gekennzeichnet (VC 20).
Folgende Abkürzungen wurden verwendet: LDM = Listings des Monats, AdM = Anwendung des Monats, SB = Simons Basic.

```
2060 -           JSR JANEIN      ; (JA/NEIN)?
2070 -           BEQ WEITER3     ; JA=>WEITER
2080 -           LDA #$6C        ; SPRUNG AUF $036C
2090 -           LDY #$03        ; VERBIEGEN
2100 -           STA START+1     ; BEI $36C STEHT
2110 -           STY START+2     ; EINE ROUTINE,
2120 -;                            DIE DEN "RUN"-
2130 -;                            BEFEHL SIMULIERT
2140 -;
2150 -;
2160 -; SYSTEMMELDUNGEN (J/N)?
2170 -; ======================
2180 -;
2190 -WEITER3    ... PRINT(TEXT5)
2200 -           JSR JANEIN      ; (JA/NEIN)?
2210 -           BNE WEITER4     ; NEIN=>WEITER
2220 -           LDA #$80        ; FLAG FUER
2230 -           STA SYSTEM+1    ; SYSTEMMELDUNGEN
2240 -;
2250 -;
2260 -; LOAD ERROR  AUSGEBEN (J/N)
2270 -; ==========================
2280 -;
2290 -;
2300 -WEITER4    ... PRINT(TEXT6)
2310 -           JSR JANEIN      ; (JA/NEIN)?
2320 -           BEQ WEITER5     ; NEIN=>WEITER
2330 -           LDA #0          ; FEHLERMELDUNGEN
2340 -           STA FEHLER+1    ; UNTERDRUECKEN
2350 -;
2360 -;
2370 -; PROGRAMMENDE
2380 -; ============
2390 -;
2400 -;
2410 -WEITER5    ... PRINT(TEXT7)
2420 -;
2430 -; VEKTOR FUER BASIC-ENDE SETZEN
2440 -; ============================
2450 -;
2460 -;
2470 -           LDA #<(MDFIKATOR)
2480 -           STA $2D         ; LOW-BYTE
2490 -           LDA #>(MDFIKATOR)
2500 -           STA $2E         ; HIGH-BYTE
2510 -           JMP READY       ; SPRUNG INS BASIC
2520 -;
2530 -;
10000-;
10010-; ASCII-TABELLEN
10020-; ==============
10030-;
10040-;
10050-TEXT1      .TX "LOADER-MAKER 64"
10060-           .BY 13,13
10070-           .TX "STARTADRESSE : "
10080-           .BY 0
10090-;
10100-TEXT2      .BY 13,13
10110-           .TX "FILENAME : "
10120-           .BY 0
10130-;
10140-TEXT3      .BY 13,13
10150-           .TX "GERAETENR. (1-9;0=UEBERNEHMEN) : "
10160-           .BY 0
10170-;
10180-TEXT4      .BY 13,13
10190-           .TX "MASCHINENPROGRAMM"
10200-           .BY 0
10210-;
10220-TEXT5      .BY 13,13
10230-           .TX "SYSTEMMELDUNGEN"
10240-           .BY 0
10250-;
10260-TEXT6      .BY 13,13
10270-           .TX "LOAD ERROR  AUSGEBEN"
10280-           .BY 0
10290-;
10300-TEXT7      .BY 13,13,18
10310-           .TX "*** LOADER GENERIERT ***"
10320-           .BY 13,13
10330-           .TX "MIT 'SAVE' SPEICHERN,"
10340-           .TX " MIT 'RUN' STARTEN"
10350-           .BY 0
10360-;
10370-TEXT8      .BY 13,13,18
10380-           .TX "*** PROGRAMMENDE ! ***"
10390-           .BY 13,13,0
10400-;
10410-TEXT9      .TX " (J/N)? "
10420-           .BY 0
10430-;
10440-;
20000-;
20010-; UNTERPROGRAMM FUER "J/N?"
20020-; =========================
20030-;
20040-;
20050-JANEIN     ... PRINT(TEXT9)
20060-           JSR BASIN       ; EINGABE HOLEN
20070-           CMP #"←"
20080-           BNE JANEIN1
20090-           PLA             ; SIEHE STAPEL-
20100-           PLA             ; MANIPULATION
20110-...PRINT(TEXT8)
20120-           JMP READY       ; SPRUNG INS BASIC
20130-JANEIN1    CMP #"J"        ; VERGLEICH MIT "J"
20140-           RTS             ; RUECKKEHR VOM
20150-;                            UNTERPROGRAMM
20160-.EN
```

Listing 30 (Schluß)

überprüft, ob die nächste Adresse schon mit der ersten Adresse nach der Endadresse ($475F), also $4760, übereinstimmt (siehe $6016 - $601D). Dieser 16-Bit-Vergleich wurde bereits im SMON vorgestellt. Bei $601E wird schließlich die Schleife beendet, falls die Abbruchbedingung (C=1) erfüllt ist.

Listing 20 ist eine Dekrementierschleife, die sich in der Wirkung nicht von Listing 19 unterscheidet. Da das Dekrementieren einer 16-Bit-Adresse beim 6510 langsamer und speicherplatzaufwendiger ist als das Inkrementieren, ist Listing 20 weniger effektiv als Listing 19.

Grundsätzlich können Sie an den Listings 19 und 20 sehen, wie man eine Typ-b-Schleife programmiert. Diese arbeitet jedoch nicht besonders schnell. Der Grund ist, daß der Bereich von $3FD2 - $475F nicht restlos in ganze Seiten (256-Byte-Blöcke)aufgeteilt werden kann. Daher sollte man sich immer überlegen, ob die Schleifendurchlaufzahl nicht auf ganze 256-Byte-Blöcke »aufgerundet« werden kann. In unserem Fall würde dies heißen, daß mit einer schnelleren Schleife der exakt 8 x 256 Byte lange Bereich $3FD2 - $47D1 invertiert wird, anstelle des »ungeraden« Bereichs $3FD2 - $475F. An einfacheren Zahlen wollen wir nun eine solche Schleife für ganze Seiten programmieren. Der 32 x 256 Byte umfassende Bereich von $2000 bis $3FFF (einschließlich) soll invertiert werden. Mit einer solcher Routine könnte das gerade sichtbare Bild bei Hi-Eddi invertiert werden.

Die einfachste Form finden Sie in Listing 21. Zuerst wird die Anfangsadresse in $14/$15 abgelegt. Ins Y-Register kommt der Wert 0. Dann wird der Wert invertiert und das Y-Register, der Low-Zähler, erhöht. Ist der Wert noch nicht 0, wird die Schleife neu durchlaufen. Andernfalls wurde gerade eine Seite abgearbeitet. Der High-Zähler ($15) wird erhöht. Ist der Inhalt des High-Zählers = $40, wird die Schleife abgebrochen. Zu bemerken ist, daß während der Schleife die Adresse $14 unverändert 0 bleibt. Die Adresse, die invertiert wird, ergibt sich folgendermaßen:

(Y+Inhalt von $14)+256*(Inhalt von $15)

Da wir auf die Adresse über das Prozessor-Register Y Einfluß nehmen können und die Adresse $14 nicht verändert werden muß, ist die Verarbeitungsgeschwindigkeit gegenüber der »Normalform« (Listing 20) gestiegen. Das High-Byte müssen wir aber weiterhin in $15 belassen. Neu führen wir den High-Zähler X ein. Im X-Register merken wir uns, wieviele Seiten invertiert werden. Diesen Wert verwenden wir als Dekrementierzähler. In unserem Fall werden $20 Seiten invertiert. Weil $20 zufälligerweise auch das High-Byte der Anfangsadresse ($2000) ist, wird dieser Wert in Listing 22 nur einmal (6005) in den Akku geladen und dann bei 6009 ins X-Register übertragen.

Beachten Sie bitte, daß in Listing 22 die Befehle »6004 TAY« und »6009 TAX« nur bei den Werten dieses Beispiels verwendet werden können. In der Regel sind eigene »LDX #«- oder »LDY #«-Befehle erforderlich. Wenn wir zum Bei-

spiel den Bereich $3FD2 - $47D1 invertieren wollen, muß die Initialisierung so aussehen:

```
LDA #D2  Low-Byte der ersten Adresse
STA 14
LDY #00  Index-Register
LDA #3F  High-Byte der ersten Adresse
STA 15
LDX #08  High-Zähler
```

... Schleife wie ab 600C in Listing 22

Damit hätten wir eine Schleife, die den Bereich #3FD2 - $475F (siehe Listings 19 und 20) invertiert und wesentlich schneller als die Listings 19 und 20 arbeitet. Da wir aber »aufgerundet« haben, wird zusätzlich der Bereich $4760 - $47D1 invertiert, obwohl wir das gar nicht wollen. Es gibt nun mehrere Möglichkeiten, dies zu verhindern:

1. Wir verwenden die Schleife aus Listing 19, müssen aber eine deutlich höhere Arbeitsdauer hinnehmen.

2. Wir verwenden die Schleife aus Listing 22 mit obiger Initialisierung. Dann invertiert eine Typ-a-Schleife den Restbereich $4760 - $47D1 ein weiteres Mal. Damit wären - eine Besonderheit der EOR #FF-Verknüpfung - im Restbereich die alten Inhalte wiederhergestellt. Diese Lösung eignet sich aber (fast) nur bei dieser logischen Verknüpfung und hilft bei den meisten anderen Typ-b-Schleifen nicht weiter.

3. Dies dürfte wohl die beste Lösung sein: Wir schreiben eine »gemischte« Schleife, die aus einer Typ-a-Schleife und einer Typ-b-Schleife besteht. Dieses Verfahren ist immer (!) möglich und wird von der BLTUC-Routine ($A3BF) des Basic-Interpreters angewandt. Diese Verschiebe-Routine zerlegt den Bereich, der verschoben werden soll, in einen Bereich der aus 256-Byte-Blöcken besteht und in einen Restbereich. Beide Bereiche werden dann getrennt verschoben.

Folgendermaßen sieht die optimale Invertierroutine für den Bereich $3FD2 - $475F aus:

a) Der exakt 7 Seiten umfassende Bereich 3FD2 - $46D1 wird mit einer Typ-b-Schleife wie in Listing 22 komplementiert.

b) Der Restbereich $46D2 - $475F wird mit einer Typ-a-Schleife wie in Listing 13 komplementiert.

Wir haben nun viele verschiedene Schleifenkonstruktionen in Theorie und Praxis behandelt. Was uns noch fehlt, sind Formeln, nach denen Sie die einzelnen Parameter (zum Beispiel den Startwert für X in einer Dekrementier-Schleife vom Typ a) errechnen können. Als Zusammenfassung finden Sie in Form von Listing 23 ein Hypra-Ass-Assemblerlisting zu mehreren Schleifenkonstruktionen. An den Quelltext-Ausdrücken können Sie sehen, wie einzelne Parameter errechnet werden können.

Merke: Sofern es der Programmablauf zuläßt, sollten Sie Inkrementierschleifen verwenden.

Bei Verschiebeschleifen ist aber oft eine Dekrementierschleife erforderlich.

Noch etwas zum Schleifen-Inhalt: Wenn mehrere Schleifen einen gleichen Innenteil haben (zum Beispiel einen Invertierbefehl), definieren Sie diesen unbedingt als Makro und nicht als Unterprogramm! JSRs sollten Sie nur beim Aufruf von ROM-Routinen verwenden.

Damit wäre das Thema »Schleifen« erst einmal abgeschlossen. Im nächsten Abschnitt (über Selbstmodifikation) werden wir uns aber wieder mit Schleifen auseinandersetzen.

```
 programm : loader-maker     0801 0a38
-----------------------------------

0801 : 0b 08 c1 07 9e 32 30 36  0a
0809 : 31 00 00 00 a2 00 86 9d  ba
0811 : a2 49 bd 1f 08 9d 3c 03  0f
0819 : ca 10 f7 4c 3c 03 a9 01  f7
0821 : a8 a2 00 20 ba ff a9 00  71
0829 : a2 5c a0 03 20 bd ff a9  c5
0831 : 00 20 d5 ff b0 03 4c 00  0b
0839 : 00 a2 1d 6c 00 03 00 00  77
0841 : 00 00 00 00 00 00 00 00  42
0849 : 00 00 00 00 00 00 86 2d  be
0851 : 84 2e 20 44 e5 a2 03 86  09
0859 : c6 bd 83 03 9d 77 02 ca  72
0861 : 10 f7 4c 74 a4 52 d5 0d  5d
0869 : 20 44 e5 a9 21 a0 09 20  d5
0871 : 1e ab 20 fd ae 20 8a ad  9e
0879 : 20 f7 b7 a6 14 a5 15 8e  37
0881 : 38 08 8d 39 08 20 cd bd  7c
0889 : a2 0f a9 00 9d 3f 08 ca  a7
0891 : 10 fa 8d 0e 08 a9 03 8d  38
0899 : 36 08 a9 a2 8d 22 08 a9  ef
08a1 : 42 a0 09 20 1e ab a2 00  44
08a9 : 20 cf ff c9 0d f0 08 9d  9e
08b1 : 3f 08 e8 e0 10 d0 f1 8e  b7
08b9 : 28 08 a9 50 a0 09 20 1e  69
08c1 : ab 20 cf ff 38 e9 30 8d  1f
08c9 : 23 08 d0 0a a9 a6 8d 22  b0
08d1 : 08 a9 ba 8d 23 08 a9 74  10
08d9 : a0 09 20 1e ab 20 1b 0a  06
08e1 : f0 0a a9 6c a0 03 8d 38  97
08e9 : 08 8c 39 08 a9 88 a0 09  fa
08f1 : 20 1e ab 20 1b 0a d0 05  5f
08f9 : a9 80 8d 0e 08 a9 9a a0  81
0901 : 09 20 1e ab 20 1b 0a f0  fc
0909 : 05 a9 00 8d 36 08 a9 b1  42
0911 : a0 09 20 1e ab a9 69 85  ba
0919 : 2d a9 08 85 2e 4c 74 a4  2e
0921 : 4c 4f 41 44 45 52 2d 4d  24
0929 : 41 4b 45 52 20 36 34 0d  4a
0931 : 0d 53 54 41 52 54 41 44  7a
0939 : 52 45 53 53 45 20 3a 20  ec
0941 : 00 0d 0d 46 49 4c 45 4e  7d
0949 : 41 4d 45 20 3a 20 00 0d  45
0951 : 0d 47 45 52 41 45 54 45  b8
0959 : 4e 52 2e 20 28 31 2d 39  93
0961 : 3b 30 3d 55 45 42 45 52  ce
0969 : 4e 45 48 4d 45 4e 29 20  c1
0971 : 3a 20 00 0d 0d 4d 41 53  44
0979 : 43 48 49 4e 45 4e 50 52  a9
0981 : 4f 47 52 41 4d 4d 00 0d  8a
0989 : 0d 53 59 53 54 45 4d 4d  40
0991 : 45 4c 44 55 4e 47 45 4e  89
0999 : 00 0d 0d 4c 4f 41 44 20  3d
09a1 : 45 52 52 4f 52 20 20 41  b7
09a9 : 55 53 47 45 42 45 4e 00  aa
09b1 : 0d 0d 12 2a 2a 2a 20 4c  1c
09b9 : 4f 41 44 45 52 20 47 45  30
09c1 : 4e 45 52 49 45 52 54 20  e8
09c9 : 2a 2a 2a 0d 0d 4d 49 54  3e
09d1 : 20 27 53 41 56 45 27 20  ee
09d9 : 53 50 45 49 43 48 45 52  ff
09e1 : 4e 2c 20 4d 49 54 20 27  fd
09e9 : 52 55 4e 27 20 53 54 41  cf
09f1 : 52 54 45 4e 00 0d 0d 12  49
09f9 : 2a 2a 2a 20 50 52 4f 47  2a
0a01 : 52 41 4d 4d 45 4e 44 45  53
0a09 : 20 21 20 2a 2a 2a 0d 0d  49
0a11 : 00 20 28 4a 2f 4e 29 3f  fd
0a19 : 20 00 a9 12 a0 0a 20 1e  fd
0a21 : ab 20 cf ff c9 5f d0 0c  c3
0a29 : 68 68 a9 f6 a0 09 20 1e  1e
0a31 : ab 4c 74 a4 c9 4a 60 5c  dd
```

Listing 31

10. Selbstmodifikation

Bevor wir uns mit dieser Programmiertechnik beschäftigen, die zwar nicht strukturiert, aber sehr trickreich ist, soll der Begriff geklärt werden.

Unter Modifikation versteht man »eine Änderung, Anpassung«. Wenn Sie bei einem Spiel einen der vielen POKE-Befehle, die im 64'er schon vorgestellt wurden, eingeben, so wird dadurch das Spiel »modifiziert«. Die Änderung ist zum Beispiel eine Erhöhung der Spielfigurenanzahl.

Selbstmodifikation bedeutet, daß ein Programm sich selbst programmgesteuert verändert. Dies wäre der Fall, wenn im Spielprogramm eine Passage stünde, die den POKE ausführt.

Wenn Sie sich für die Selbstmodifikation von Basic-Programmen interessieren, finden Sie in der Zeitschrift »Happy-Computer« (Ausgabe 8/85) unter der Überschrift »Lernen Sie Ihren Commodore 64 kennen« alles, was Sie wis-

sen müssen. Auf simulierten Direktmodus wurde im 64'er schon mehrfach eingegangen, unter anderem in der »Memory Map mit Wandervorschlägen«.

Wir werden uns an dieser Stelle ausschließlich mit der Selbstmodifikation von Maschinenprogrammen befassen. Als erstes Beispiel nehmen wir Listing 24.

Es handelt sich um eine selbstmodifizierende Schleife, die den Bereich $2000 - $3FFF komplementiert.

TRACEn Sie doch einmal Listing 24 mit dem SMON und vergleichen Sie die disassemblierten Befehle mit den ursprünglichen Werten, die Sie in Listing 24 finden. Sie werden erkennen, daß die Befehle »6002 LDA 2000,Y« und »6007 STA 2000,Y« aufgrund der INC-Befehle immer auf andere Adressen zugreifen. Besagte INC-Befehle erhöhen jeweils das High-Byte des Operanden. Ist dieses schon $40, so wird die Schleife beendet. In Listing 25 sehen Sie, wie unsere Schleife aus Listing 24 aussieht, wenn sie fertig durchlaufen wurde. Ein weiterer Start bewirkt, daß das Programm sich früher oder später selbst invertiert und darum abstürzt.

Was nämlich unserem Listing 24 fehlt, damit es mehr als einmal arbeitet, ist eine Initialisierung, die jedesmal den Ausgangswert ($2000) in die LDA/STA-Befehle einsetzt. In Listing 26 sehen Sie eine solche Initialisierung (6000 - 600F). Die Adresse $FFFF (bei 6012 und 6017) ist ein Dummy-Wert, das heißt er dient nur zum vorläufigen Ausfüllen von Adressen und hat keine programmtechnische Bedeutung. Der Dummy-Wert wird ohnehin von der Initialisierung überschrieben; wir hätten also statt $FFFF auch $040C oder andere verwenden können. Wichtig ist nur, daß »LDA Dummy,Y« 3 Byte belegt.

Ein besonderer Vorteil der Selbstmodifikation ist es, daß selbstmodifizierende Schleifen keine Zähler in der Zeropage benötigen, weil der Zähler praktisch im Programm selbst liegt. In puncto Geschwindigkeit sind selbstmodifizierende Schleifen den herkömmlichen aber oft unterlegen.

Ein weiterer Vorteil von ihnen ist aber, daß man außer mit weniger Zeropage-Speicherplätzen auch mit weniger Registern auskommen kann (sofern man hier Einsparungen vornehmen will). Listing 27 beispielsweise invertiert den Bereich $3FD2 - $475F. X- und Y-Register sowie die Zeropage bleiben unverändert, lediglich der Akkumulator fungiert als Arbeitsregister.

Listing 28 kopiert den Basic-Interpreter ($A000 - $BFFF) ins RAM an gleicher Adresse, wobei nur das X-Register verwendet wird (!).

Nun wollen wir sehen, wie man bei der Entwicklung selbstmodifizierender Programme unter Zuhilfenahme eines guten Assemblers (Hypra-Ass) vorgehen muß.

Zunächst einmal müssen diejenigen Stellen, an denen Modifikationen vorgenommen werden, mit Label definiert werden. Von diesen Label aus können die Stellen im Speicher die geändert werden sollen, leicht berechnet werden.

Befehlscode	= LABEL + 0 =	LABEL
Low-Operand	=	LABEL + 1
High-Operand	=	LABEL + 2

Bei 2-Byte-Befehlen wird der Parameter wie der Low-Operand eines 3-Byte-Befehls errechnet.

Als Beispiel finden Sie in Form von Listing 29 einen Quelltext (Assembler: Hypra-Ass) für Listing 28. Während in Listing 28 der Ausgangswert bei 6010 »LDX 0000« und bei 6013 »STX 0000« ist, wurde im Quelltext $FFFF verwendet (270, 280), um den Assembler zu zwingen, den Dummy-Wert als 16-Bit-Adresse abzulegen (und nicht als Zeropage-Adresse, wodurch der Befehl nur 2 statt 3 Byte belegen würde).

Die Stellen, die modifiziert werden, wurden mit »MOD1« und »MOD2« definiert. MOD1 ist zugleich der Schleifenbeginn.

Nachdem Sie jetzt den Eingang gefunden haben, möchte ich einige Anregungen liefern, wie Sie die Vorteile der Selbstmodifikation nutzen können. Wir werden hier die Anwendung nach den verschiedenen Adressierungsarten unterteilen.

a) Anwendung auf absolute Adressierung

Bei der Stapelmanipulation haben wir schon ein Verfahren kennengelernt, den Befehl JSR (indirekt), der im normalen 6510-Befehlssatz nicht existiert, zu simulieren.

Folgendermaßen kann über Selbstmodifikation ein Unterprogramm ab ADRESSE aufgerufen werden.

```
                LDA # <ADRESSE
                STA SPRUNGBEFEHL+1 ; Low-Operand
                LDA # >ADRESSE
                STA SPRUNGBEFEHL+2 ;
                High-Operand
SPRUNGBEFEHL
                JSR $FFFF ; $FFFF=Dummy
```

Genauso kann man mit dem JMP-Befehl verfahren. Sogar bei den Schieber-, Dekrementier- und Inkrementierbefehlen, die im Gegensatz zu JMP die indirekte Adressierung nicht haben, ist auf diese Weise eine Simulation der indirekten Adressierung möglich.

Wird eine Sprungtabelle per Selbstmodifikation verarbeitet, müssen die Sprungadressen in der Tabelle nicht (!) dekrementiert werden.

b) Anwendung auf Immediate-Befehle

Oft müssen Werte, die berechnet werden, auf dem Stapel oder im Speicher abgelegt und dann, wenn sie gebraucht werden, wieder aufgenommen werden.

Ein Beispiel hierfür ist der »Basic-Start-Generator« (64'er, 7/85, Seite 74). Bei Erwähnung dieses Programms taucht natürlich die Frage auf, ob es sich hier noch um ein selbstmodifizierendes Programm handelt oder ob der »Basic-Start-Generator« nicht eher zu den Programmgeneratoren zählt. Diese Frage ist voll berechtigt. Deshalb wollen wir darauf kurz eingehen.

Der »Basic-Start-Generator« ist eindeutig den Programmgeneratoren zuzuordnen, da der generierte Programmteil nie angesprungen wird und somit ein eigenständiges Programm darstellt. Das Programm modifiziert also nicht sich selbst, sondern vielmehr ein zweites Programm, welches dann vom Benutzer gespeichert werden kann.

Die Programmierung ist aber bei Programmgeneratoren nicht anders als bei selbstmodifizierenden Programmen. Auf den Unterschied Programmgeneration/Selbstmodifikation werden wir an späterer Stelle näher eingehen.

Zunächst wollen wir aber ein praktisches Beispiel für die Anwendung der Modifikation von Immediate-Befehlen behandeln. Oft steht man vor dem Problem, ein Register zu sichern und später wieder zu holen. Im Falle des Akkumulators sieht das so aus:

```
        PHA         ; Akku sichern
        .....       ; weiteres Programm
        PLA         ; Akku wieder holen
```

Beim X-Register wird's schon ungünstiger:

```
        TXA         ; X-Register in Akku
        PHA         ; Akku sichern
        ......      ; weiteres Programm
        PLA         ; Akku wieder holen
        TAX         ; Akku ins X-Register
```

Hier wird also zusätzlich der Akku beeinflußt. Wenn dies vermieden werden muß, wird folgender Weg gewählt:

```
        STX $02     ; $02 = Zwischenspeicher
        ....        ; weiteres Programm
        LDX $02     ; X wieder holen
```

Für die Sicherung des X-Registers gibt es aber noch eine weitere Lösung, die den X-Wert im Programm ablegt und

dadurch nicht den Stapel oder einen Zwischenspeicher außerhalb des Programms benötigt.

```
        STX GETX+1   ; X direkt in Immediate-Befehl
                       schreiben
        ....         ; weiteres Programm
GETX    LDX #$00     ; $00 = Dummy-Wert
```

Obiges Beispiel kann sehr leicht auf Akkumulator oder Y-Register umgeschrieben werden.

Folgendermaßen kann das X-Register mit dem Akkumulator verglichen werden:

```
        STX VGL+1    ; in Vergleichsbefehl ablegen
        (.....       ; evtl. weitere Programme)
VGL     CMP #$00     ; $00 = Dummy
```

Als letztes Beispiel für die Anwendung auf Immediate-Befehle soll das Y-Register zum Akkumulator addiert werden:

```
        STY ADD+1    ; in Arithmetikbefehl ablegen
        (.....       ; evtl. weiteres Programm)
        CLC          ; Carry vor Addition
ADD     ADC #$FF     ; $FF = Dummy
```

Die Anwendungsmöglichkeiten sind hier unbegrenzt.

c) Anwendung auf komplette Befehle

Bisher haben wir nur die Parameter einzelner Befehle modifiziert. Es ist selbstverständlich auch möglich, die Befehlscodes oder die kompletten Befehle zu modifizieren.

Wenn nur der Befehlscode geändert wird (zum Beispiel ein ORA #- in einen EOR #-Befehl) bleiben die Parameter erhalten. Es könnte ferner ein impliziter Befehl (SEI,CLI,CLD, DEX,INX,....) geändert werden, um beispielsweise zwischen In- und Dekrementieren umzuschalten. Außerdem könnte bei einem BRANCH-Befehl die Sprungbedingung (CS,CC,VS, VC,NE,EQ) geändert werden. Aus BCS könnte also leicht BCC werden.

Weil man hier die Opcodes der Befehle kennen muß, empfehle ich das erste 64'er Extra (Ausgabe 9/85) oder die Tabelle am Ende dieser Ausgabe.

Nun lösen wir noch das häufig auftretende Problem, wie die Ausführung eines Unterprogramms verhindert wird. Dazu werden wir drei Lösungen (I - III) entwickeln.

I. Die Adresse FLAG wird auf 0 gesetzt, wenn das Unterprogramm ausgeführt werden soll; auf einen anderen Wert, wenn es nicht ausgeführt werden soll.

```
        LDA #0       ; Flag für Ausführung
        STA FLAG     ; Flag setzen
        (.....       ; evtl. weiteres Programm)
        LDA FLAG     ; Flag testen
        BNE NEIN     ; Flag <> 0, also nicht ausführen
        JSR UNTER-
        PROGRAMM     ; Aufruf
NEIN    .....          weiteres Programm
```

Das Flag könnte auch am Beginn des Unterprogramms abgefragt und dann (wenn FLAG <> 0) das Unterprogramm verlassen werden.

II. Als ersten Befehl des Unterprogramms verwenden wir NOP:

```
UP      NOP          ; Beginn des Unterprogramms
        .....        ; Fortsetzung des Unterprogramms
```

So wird die Ausführung des Unterprogramms gestattet:

```
        LDA #$EA     ; Opcode für NOP
        STA UP       ; an Anfang des Unterprogramms
                       schreiben
```

Und so wird sie verhindert:

```
        LDA #$60     ; Opcode für RTS
        STA UP       ; an Anfang des Unterprogramms
                       schreiben
```

Wer noch einen NOP-Befehl und damit 1 Byte sparen möchte, kann den NOP-Befehl entfallen lassen. Dann muß auch der Opcode $EA beim Erlauben des Unterprogramms in den Opcode des ersten Byte im Unterprogramm geändert werden. Weil dies ziemlich mühselig ist, ziehe ich die ursprüngliche Lösung II trotz des um 1 Byte erhöhten Speicherbedarfs vor.

III. Das beste Verfahren. Wir schalten den JSR-Befehl aus, indem wir ihn in einen BIT-Befehl abändern.

```
AUFRUF  JSR Unterprogramm
JSR ausschalten:
        LDA #$2C     ; Opcode für BIT
        STA AUFRUF
JSR wieder erlauben:
        LDA #$20     ; Opcode für JSR
        STA AUFRUF
```

Der JSR-Opcode kann auch mit $0C überschrieben werden. $0C ist ein illegaler Opcode für ein 3-Byte-NOP und arbeitet mit allen mir bekannten Versionen des C 64. Ob er ebenfalls auf dem C 128 läuft, konnte ich noch nicht prüfen.

Im übrigen können mit dem soeben beschriebenen Verfahren auch andere Befehle ausgeschaltet werden, zum Beispiel JMP, LDA, STA und so weiter. Wenn aber der JSR-Opcode mit $2C (BIT) überschrieben wird, ist darauf zu achten, daß bei der Ausführung des BIT-Befehls die Prozessorflags gesetzt werden.

Sicherlich gibt es noch mehr Problemlösungen als I - III, aber III dürfte wohl kaum zu übertreffen sein.

d) Anwendung auf mehrere Befehle

Selbstverständlich können ganze Befehlsfolgen, also größere Programmteile gegeneinander ausgetauscht werden. Zu beachten ist nur, daß die Routinen, die gegeneinander ausgetauscht werden, auch in dem Bereich, in den sie vom Programm aus geschrieben werden, lauffähig sind. Dies ist vor allem dann gegeben, wenn nur die relative Adressierung verwendet wird und dadurch die Routine im Speicher frei verschoben werden kann.

e) Anwendung auf Tabellen

Dieser Anwendungsfall würde auch zum Abschnitt über »Tabellen« passen.

Wir bleiben hier bei der Theorie, denn die Umsetzung in ein Programm ist nicht mehr schwer. Vielmehr soll Ihre Kreativität nicht durch Unmengen von Beispielen gehemmt werden.

Zunächst wollen wir uns ein wenig mit dem SMON befassen. Wenn Sie den Disk-Monitor einschalten, kopiert das Programm einen Floppy-Befehl («U1 ..«) vom Ende des SMON in einen Bereich zwischen $02A0 und $02FF. Dieser Lesebefehl wird nach Bedarf modifiziert, zum Beispiel wird beim Schreiben der »U1«- in einen »U2«-Befehl umgewandelt oder die Angabe des einzulesenden Blocks wird geändert. Dies wäre ein typisches Anwendungsbeispiel für Selbstmodifikation, wenn der Lesebefehl nicht erst in einen Bereich außerhalb des Programms kopiert würde (worin ich keinen Sinn sehe), sondern am Ende des SMON (etwa bei $CFF0) bliebe und dort modifiziert würde.

Im Hi-Eddi liegt eine Tabelle, die die High-Byte der Bit-Map-Anfangsadressen beinhaltet. Diese Tabelle wird von Hi-Eddi bei jedem Bildwechsel umgerechnet.

Nach den vorausgegangenen zwei Beispielen an Spitzenprogrammen aus dem 64'er möchte ich noch andere Anwendungsbeispiele nennen.

Besonders flexible Programme erlauben Eingriffe des Anwenders in die Befehls- oder Text-Tabellen. So können Bildschirmmasken editiert oder Eingabemasken erstellt werden.

Ein solches Programm braucht sich nach den Modifikationen nur selbst abzuspeichern. Weil hier unter Umständen ein erheblicher Teil des Programmschutzes verlorengeht, werden dann lediglich die Tabellen gespeichert.

Ein Adventure-Generator modifiziert in der Regel auch nur die Tabellen eines fertigen Adventureprogramms, das eigentliche Programm bleibt unverändert. In diesen Tabellen sind die einzelnen Spielsituationen enthalten.

Bei diesen (theoretischen) Fällen wollen wir es belassen. Letztendlich muß ja der Programmierer entscheiden, inwieweit er die Selbstmodifikation auf Tabellen anwenden kann.

f) Das Beispielprogramm »Loader-Maker 64«

Wie aus dem Namen des Beispielprogramms schon zu entnehmen ist, handelt es sich um einen Programmgenerator. Da - wie gesagt - die Programmierung wie bei selbstmodifizierenden Programmen ist, habe ich bewußt einen Programmgenerator als Beispiel gewählt.

Als Listing 31 finden Sie ein MSE-Listing, falls Sie »Loader-Maker 64« bequem abtippen wollen und an der Anwendung des Programms interessiert sind. Deshalb zunächst eine Kurzbeschreibung für Anwender.

»Loader-Maker« ermöglicht es Ihnen, zu einem Programm ein (Maschinensprache-) Ladeprogramm zu generieren, welches normal geladen und mit »RUN« gestartet wird, worauf es das nachzuladende Programm nachlädt und startet.

Nach dem Laden von »Loader-Maker« wird dieses Programm durch SYS 2154,START gestartet. START ist eine Variable und wird durch die Startadresse des nachzuladenden Programms ausgedrückt. Soll ein Basic-Programm nachgeladen werden, hat diese Adresse keine Bedeutung (einfach SYS2154,0 eingeben). Bei einem Maschinenprogramm handelt es sich hier um die Adresse, mit der das Programm über »SYS« gestartet wird (49152 beim SMON $C000).

Das Programm meldet sich mit »Loader-Maker 64« und gibt die Startadresse aus. Dazu können Sie den Filenamen eingeben.

Bei allen weiteren Eingaben (Gerätenummer, von der geladen werden soll; Maschinenprogramm j/n; Systemmeldungen wie »SEARCHING FOR« ausgeben j/n; LOAD ERROR bei Ladefehler ausgeben j/n) können Sie das Programm durch Eingabe des Linkspfeils abbrechen. Sind alle Eingaben gemacht worden, kommt die Meldung »LOADER GENERIERT« und der Lader kann mit »SAVE« gespeichert werden.

Wenn das nachzuladende Programm von der Adresse geladen werden soll, von der auch das Ladeprogramm selbst eingelesen wurde, ist als Gerätenummer nur 0 einzugeben.

Befassen wir uns nun mit dem Programm, dessen Quelltext Sie als Listing 30 finden.

Die Zeilen bis 990 stellen das Ladeprogramm in unmodifizierter Form dar und enthalten viele Dummywerte, wie zum Beispiel die (unsinnige) Startadresse 0 in Zeile 820.

Mit 1000 beginnt die Modifikationsroutine. Nach 1120 wurde die Startadresse eingelesen, die ja per SYS übergeben wurde, und wird wieder mit dem Titel ausgegeben. 1100/1110 schreiben die Startadresse hinter den JMP-Befehl in Zeile 820.

1150 - 1350 bringen das (noch unmodifizierte) Gerüst in den Ausgangszustand, der dann nach Bedarf geändert wird.

1400 - 1550 holen den Filenamen, legen ihn bei NAME (850) ab, berechnen gleich die Länge des Filenamens und legen diese bei LAENGE (750) ab.

1600 - 1720 holen die Geräteadresse. Da diese im ASCII-Format vorliegt, muß der ASCII-Code von 0 abgezogen werden (1640/1650). Wurde 0 eingegeben, wird der LDX #DEVICE-Befehl (730) in »LDX $BA« geändert. Die Adresse $BA enthält jeweils die Adresse, von der das letzte Programm geladen wurde.

1750 - 1850 fragen, ob das nachzuladende Programm mit der per SYS übermittelten Startadresse gestartet wird (Eingabe »j«). Wurde »n« eingegeben, muß das Programm über den Basic-Befehl RUN eingegeben werden. Auf eine entsprechende Routine (870 - 980) wird die Startadresse gestellt (1810 - 1840).

1900 - 1970 ermöglichen die Einstellung, ob »SEARCHING..«, »LOADING« etc. ausgegeben werden sollen.

Soll im Falle eines Ladefehlers das Programm nicht gestartet und stattdessen »LOAD ERROR..« ausgegeben werden, wird dies bei 2000 - 2090 festgelegt. Wird die Fehlerausgabe unterdrückt, muß der BCS-Befehl (810) unschädlich gemacht werden. Dies geschieht einfach dadurch, daß die Sprungweite auf 0 gesetzt wird (2070/2080).

Am Programmende wird noch eine Meldung ausgegeben (2140 - 2160) und der Vektor für das Ende des Basic-Programms neu gesetzt, damit das generierte Ladeprogramm mit »SAVE« gespeichert werden kann.

10000 - 10310 enthalten nur die Text-Tabellen.

Von 15000 bis zur letzten Zeile (15170) steht ein Unterprogramm, daß bei jeder J/N-Entscheidung über »JSR J,N« aufgerufen wird.

Es gibt den Text »(J/N)?« aus (15030 - 15050) und holt eine Eingabe. Ist diese »J«, so ist nach dem Verlassen des Unterprogramms (1517) das Zero-Flag gesetzt (andernfalls nicht).

Wurde der Linkspfeil eingegeben, wird das Programm abgebrochen und eine entsprechende Meldung ausgegeben (15100 - 15150).

Wie wir nun gesehen haben, handelt es sich bei »Loader-Maker« um einen Programmgenerator. Mit zwei kleinen Änderungen wird er jedoch zum selbstmodifizierenden Ladeprogramm. Wir müssen nur die beiden »JMP READY.«-Befehle (2240/15150) in »JMP SYSTEM« umwandeln, wodurch am Programmende der generierte Lader angesprungen würde. Schon hätten wir ein selbstmodifizierendes Ladeprogramm.

Um Ihnen noch die Anwendung des Loader-Maker zu erleichtern, hier zwei Eingabebeispiele:

```
Startadresse.....49152
Filename.....SMON $C000
Geräteadresse.....0
Maschinenprogramm .....j
Systemmeldungen.....j
LOAD ERROR ausgeben ..j

Startadresse.....0 (bedeutungslos)
Filename.....HI-EDDI
Geräteadresse.....8
Maschinenprogramm .....n
Systemmeldungen.....n
LOAD ERROR ausgeben ..j
```

g) Verbesserungen an »Tabellen-Beispiel«

Zum Abschluß des Themas »Selbstmodifikation« wollen wir noch kleine Verbesserungen am Programm »Tabellen-Beispiel« erwähnen. Ich werde hier eher Anregungen geben als fertige Änderungsvorschläge.

Zunächst soll die Adresse XSAVE (zum Sichern des X-Registers in Schleifen) überflüssig werden. So könnte es nun gesichert werden:

```
XSV   STX  GETX
.....
GETX  LDX  #$00      0=Dummy; hier wird X wieder
                       aufgenommen.
```

Auch die Sprungtabelle läßt sich - viel einfacher, finde ich - anders handhaben:

```
      LDA  J?LO,X    JMLO oder JELO
      STA  SPRO+1
      LDA  J?HI,X    JMHI oder JEHI
      STA  SPRG+2
SPRG  JMP  0000
```

In den Tabellen JMLO/JMHI und JELO/JEHI (Low- und High-Bytes der Sprungadressen) dürfen die Adressen aber nicht dekrementiert werden.

Wird ein JSR (IND)-Befehl simuliert, muß nach wie vor die Rücksprungadresse auf den Stapel gelegt werden. Dies würde entfallen, wenn die Rücksprungadresse direkt auf »SPRG JMP 0000« folgen und der JMP-Befehl bei SPRG in JSR umgewandelt würde.

64'er PROGRAMM-SERVICE

Bestellungen aus anderen Ländern bitte per Auslandspostanweisung! Achtung: Nicht die eingeheftete Zahlkarte verwenden!

Bestellungen aus der Schweiz bitte direkt an: Markt&Technik Vertriebs AG, Kollerstr. 3, CH-6300 Zug, Tel. 042/41 56 56. Bestellungen aus Österreich bitte direkt an: Bücherzentrum Meidling, Schönbrunnerstr. 261, 1120 Wien, Tel. 02 22/83 31 96. Mikrocomput-ique Erhard Schiller Fasangasse 21, 1030 Wien, Tel. 02 22/78 56 61.

Programme aus den früheren Ausgaben

Sonderheft: Professionelle Anwendungen
2 Disketten
Bestell-Nr. L6 85 S7D **DM 34,90***
4 Kassetten
Bestell-Nr. L6 85 S7K **DM 34,90***

Sonderheft: Top-Themen
2 Disketten
Bestell-Nr. L6 85 S6 **DM 34,90***

Sonderheft: Floppy, Datasette
Diskette
Bestell-Nr. L6 85 S5D **DM 29,90***
Kassette
Bestell-Nr. L6 85 S5K **DM 19,90***

Sonderheft: Grafik
Bestell-Nr. L6 85 S4A **DM 29,90***

Sonderheft: Spiele
Beide Disketten in einem Paket!
Verwenden Sie **nur** diese Bestell-Nr.:
Bestell-Nr. L6 85 S3A **DM 34,90***

Sonderheft: Abenteuerspiele
Bestell-Nr. L6 85 S2 **DM 34,90***

Sonderheft: Tips & Tricks
(2. ü. Auflage)
Floppy-Utilities CB 023 **DM 29,90***
Hilfsprogramme CB 024 **DM 29,90***

Ausgabe 12/85
Bestell-Nr. L6 85 12A **DM 29,90***

Ausgabe 11/85
Bestell-Nr. L6 85 11A **DM 29,90***
Commodore 64

Ausgabe 10/85
Leider hat sich in die Bestell-Nummer der letzten Programm-Service-Anzeige ein Druckfehler eingeschlichen. Die korrigierte **Bestell-Nummer** lautet:
L6 85 10A **DM 29,90***
Commodore 64

Ausgabe 9/85
Bestell-Nr. L6 85 09A **DM 29,90***
Commodore 64

Ausgabe 8/85
Bestell-Nr. L6 85 08A **DM 29,90***
Commodore 64

Ausgabe 7/85
Bestell-Nr. L6 85 07A **DM 29,90***
Commodore 64

Ausgabe 6/85
Bestell-Nr. L6 85 06A **DM 29,90***
Commodore 64

Ausgabe 5/85
Bestell-Nr. L6 85 05A **DM 29,90***
Commodore 64

Ausgabe 4/85
Bestell-Nr. L6 85 04A **DM 29,90***

Ausgabe 3/85
Bestell-Nr. L6 85 03A **DM 29,90***

Ausgabe 2/85
Bestell-Nr. L6 85 02A **DM 29,90***

Ausgabe 1/85
Bestell-Nr. L6 85 01A **DM 29,90***

Ausgabe 12/84
Bestell-Nr. CB 022 **DM 29,90***

Ausgabe 11/84
Bestell-Nr. CB 020 **DM 29,90***

Ausgabe 10/84
Bestell-Nr. CB 019 **DM 29,90***

Bedeutung der Abkürzungen

*LdM = Listing des Monats
*AdM = Anwendung des Monats
*SB = Simons Basic
*GV = Grundversion
*GV> = alle Speicherversionen können verwendet werden (einschließlich GV)
*3K = 3-KByte-Speichererweiterung wird benötigt
*8K> = Speichererweiterung größer als 8 KByte wird benötigt
*UPB = Unterprogrammbibliothek

* Alle Preise inklusive Mehrwertsteuer.

Bitte verwenden Sie für Ihre Bestellung **nur** die eingeheftete Postscheck-Zahlkarte zur Überweisung des Rechnungsbetrags.

Fehlende Hefte erhalten Sie bei: Markt & Technik Vertrieb 64'er Hans-Pinsel-Str. 2 8013 Haar

Damit soll das Thema »Selbstmodifikation« abgeschlossen sein. Die vorgestellten Programmiertechniken bieten fast unbegrenzte Möglichkeiten, hier konnte ich nur einen kleinen Überblick geben, welcher aber für fortgeschrittene Programmierer ausreicht.

11. Mehr über relative Adressierung

So wie wir schon die Tücken der Zeropage-Adressierung zumindest teilweise beseitigen konnten, wollen wir uns mit der in vergleichbarer Weise leistungsstarken Relativ-Adressierung auseinandersetzen.

a) So vermeidet man JMP

Oft muß eine Stelle im Programm angesprungen werden, ohne daß erst eine Bedingung geprüft wird. Diese Stelle ist nicht selten weniger als 128 Byte vom Sprungbefehl entfernt, könnte also relativ adressiert werden.

Dennoch ist es in vielen Fällen möglich, einen Branch-Befehl – obwohl diese Befehle eine Bedingung (C=0..) prüfen – zu verwenden.

Beispiel:

```
7050    BNE 7040
7052    JMP 708A
```

Kann ersetzt werden durch:

```
7050    BNE 7040
7052    BEQ 708A
```

da bei 7052 in jedem Fall das Z-Flag = 0 ist (dafür sorgt der Abfang-Befehl BNE) und somit immer verzweigt wird.

Man könnte den BEQ-Befehl als »Pseudo-Verzweigungsbefehl« bezeichnen, da die Bedingung gar nicht überprüft werden müßte (sie ist sowieso erfüllt).

Der Branch-Befehl übertrifft den JMP-Befehl deutlich an Effektivität, da ein Byte weniger verbraucht wird.

Im übrigen ist auch bei

```
7050    BVS 7040
7052    CLV
```

der CLV-Befehl überflüssig, solange vor 7052 der Befehl von 7050 verarbeitet wird.

b) Zugriff auf Befehle in »Umgebung«

Unter »Umgebung« wollen wir den Bereich um einen Programmteil verstehen, der über relative Adressierung angesprochen werden kann. Da in diesem oft ähnliche Befehlsfolgen stehen wie im anderen Programm, läßt sich hier durch gezielten Zugriff auf die »Umgebung« der Speicherplatzbedarf senken.

Beispielsweise stehen an vielen Stellen im Programm RTS-Befehle. Diese werden, wenn ein Unterprogramm verlassen werden soll, manchmal durch einen Branch-Befehl angesprungen.

```
X1      RTS         ; Ende eines im Speicher voraus-
                      gehenden Unterprogramms
UP      .....       ; Unterprogramm
TEST    BEQ X2      ; Unterprogramm verlassen, falls Z=0
        .....       ; andernfalls weiteres Programm
X2      RTS         ; Ende des Unterprogramms
```

Wenn X1 von TEST aus relativ adressiert werden kann, können wir folgendermaßen ein Byte sparen:

```
X1      RTS
UP      .....
TEST    BEQ X1      ; nach X1 springen, wo auch ein RTS
                      steht
        .....
X2      RTS         ; wird nicht mehr benötigt
```

Noch ein Beispiel aus dem Basic-Interpreter. Bei Adresse $AF08 stehen zwei Befehle, die einen SYNTAX ERROR erzeugen.

Nun gibt es im Basic-Interpreter unzählige Stellen, an denen ein SYNTAX ERROR aufgerufen werden muß. Deshalb steht dort nur »JMP $AF08«. Diese Stellen werden bei Bedarf relativ adressiert, so daß nicht an jeder Stelle, an der ein SYNTAX ERROR aufgerufen wird, der Befehl »JMP $AF08« stehen muß.

Zur Übung könnten Sie noch versuchen, im Programm Tabellen-Beispiel (Listing 11) die Menüroutine (insbesondere die Routinen HOME, DOWN, UP, EXEC), in der beispielsweise wiederholt STX MPT steht, durch Zugriff auf »Umgebung« zu optimieren. Besonders hilfreich dürfte es sein, zunächst statt Branch-Befehlen JMPs einzusetzen und dann zu überlegen, inwieweit die JMPs durch Branches ersetzt werden können, weil zum Beispiel nach »LDX # 0« das Z-Flag immer gesetzt ist etc.

12. Puffer-Technik

In der Computerei fällt der Begriff »Puffer« sehr häufig. Beim C 64 gehören der Kassetten- und der Tastaturpuffer gemeinhin zu den bekanntesten Puffern. Statt »Puffer« kann man auch Zwischenspeicher sagen. Puffer dienen nämlich immer als Zwischenspeicher.

Zunächst wollen wir klären, was zu einem Puffer gehört.

a) Was benötigt ein Puffer?

– Pufferspeicher

Selbstverständlich muß ein Puffer einen bestimmten Speicherbereich belegen, in dem die Werte zwischengespeichert werden.

Ebenso muß die maximale Puffergröße festgelegt werden, damit geprüft werden kann, ob sich der Puffer schon angefüllt hat. Beim Kassettenzugriff werden vorerst alle Byte, die auf die Kassette sollen, im Puffer (ab $033C) zwischengespeichert. Ist dieser Puffer voll, würde er beim nächsten Byte, das er aufnehmen soll, überlaufen (das heißt, die maximale Puffergröße überschreiten). Deshalb wird dann Byte für Byte der Puffer entleert, indem die Bytes auf Kassette geschrieben werden. Jedes Byte, das auf Kassette geschrieben wurde, belegt keinen Speicher mehr im Puffer, so daß der Puffer wieder aufnahmefähig ist.

Damit das Programm, das den Puffer verwaltet, auch weiß, aus welcher Adresse im Puffer es sich das nächste Byte holen soll beziehungsweise wo im Puffer das nächste Byte abgelegt werden soll, gibt es noch einen

– Pufferzeiger

Auf englisch heißt er »BUFFER-POINTER«, woher auch die Abkürzung »B-P« beim Floppy-Befehl zur Manipulation des Pufferzeigers stammt.

Dieser Pufferzeiger kann mit dem Stapelzeiger verglichen werden. Auf keinen Fall ist er mit dem

– Puffervektor

zu verwechseln, der die Startadresse des Pufferspeichers beinhaltet. Ein Puffervektor ist nicht unbedingt erforderlich, erhöht aber die Flexibilität.

Damit wären die Fachausdrücke im Zusammenhang mit Puffern geklärt.

b) Wann verwendet man Puffer?

Puffer dienen in der Regel als Zwischenspeicher, wie zum Beispiel der Basic-Eingabepuffer (ab $0200).

Im Fall des Tastatur- oder Diskettenpuffers aber sind die Puffer als Verbindungsstelle zwischen zwei parallel arbeitenden Programmen beziehungsweise Peripheriegeräten vorgesehen (interruptgesteuerte Tastaturabfrage/Hauptprogramm im Computer, DOS/Betriebssystem des Computers).

Die Puffer sind in diesen Fällen ein Bereich, auf den zwei (quasi-) parallel arbeitende Programme zugreifen.

Bei Computern, die ein wirklich starkes Multitasking bieten (wie der Commodore Amiga) finden Puffer weitaus mehr Verwendung als beim C 64, der nur einen quasiparallelen Ablauf ermöglicht.

Daher werden bei ihm Puffer hauptsächlich im I/O-Bereich verwendet, zum Beispiel bei Druckern, Datasette, Floppy, Tastatur etc. (I/O = Input/Output = Eingabe/Ausgabe).

13. Pass-Technik

a) Begriffserläuterung

Der Begriff »Pass« wurde schon mehrfach im 64'er erläutert (unter anderem Ausgabe 7/85, Seite 51).

Am einfachsten kann der Begriff als »Schritt beim Programmmablauf« verstanden werden. Mit »Schritt« ist hier nicht ein einzelner Befehl, sondern ein größerer Block im Programm gemeint.

Wenn ein Programm in 3 Passes (Durchläufen) arbeitet, heißt dies, daß 3 Schleifen hintereinander abgearbeitet werden, die alle eine Teilaufgabe erfüllen, die in Verbindung mit den anderen Passes erst eine größere Aufgabe (zum Beispiel eine Assemblierung) ausfüllen kann. Jeder einzelne Pass führt eine bestimmte Tätigkeit aus, die für das Funktionieren der darauffolgenden Passes unbedingt erforderlich ist. Pass 1 wirkt also wie eine Initialisierung von Pass 2 etc.

Komplexe Programme in Schritte (Passes) zu gliedern, gehört zu den Grundregeln des strukturierten Programmierens.

b) Beispiele von Anwendungen der Pass-Technik

Besonders umfangreiche Programme wie Assembler (Hypra-Ass), Compiler (Austro-Speed) und Interpreter (Comal) sind immer in mehrere Passes eingeteilt.

So erfolgt bei den meisten Assemblern im ersten Pass ein Syntax-Check und das Anlegen der Symbol-Tabelle. Erst im zweiten Pass wird der Objektcode generiert, wobei die bereits erstellte Symboltabelle benötigt wird.

14. Diverse Tips zur optimalen Speichernutzung

Mit übermäßig viel RAM ist der C 64 bestimmt nicht gesegnet. Bei vielen Anwendungen (zum Beispiel Datenverarbeitung) braucht man auch das letzte Byte.

Sie werden nun mehrere Tips erhalten, wie man den wenigen vorhandenen Speicher möglichst sparsam verwenden kann.

Zu den speicherplatzaufwendigsten Einrichtungen gehören die Puffer. Der Kassettenpuffer beispielsweise belegt den RAM-Bereich $033C - $03FB, auf den man somit oft verzichten muß.

Hier wollen wir einfach den Kassettenpuffer in den Bildschirmspeicher (ab $0400 in Normaleinstellung) verlegen.

```
LDA #<$400
LDY #>$400
STA $B2
STY $B3
```

Da der Bildschirm beim Kassettenbetrieb ohnehin abgeschaltet wird, fällt dies nicht auf. Nach dem Kassettenbetrieb sollte man aber den Bildschirm unverzüglich löschen.

Ebenso kann man andere Puffer, für die es einen Vektor gibt, problemlos nach $400 verlegen, sofern sie nicht größer als 1000 Byte sind.

Ein Problem für sich stellt das RAM ab $E000 (also unter dem Betriebssystem!) dar. Diesen Speicher kann man nur durch Bank-Switching nutzen, wobei man noch auf das Betriebssystem verzichten muß, solange der $E000-Bereich auf RAM geschaltet ist.

Hier können wir uns zunutze machen, daß der VIC auch ohne Ändern des Prozessor-Ports (Adresse $0001) auf diesen RAM-Bereich zugreifen kann. Für Grafikbilder oder einen geänderten Zeichensatz ist der $E000-Bereich bestens geeignet.

Oft wird der $E000-Bereich zur Ablage verschiedener Daten verwendet, auf die nicht andauernd zugegriffen werden muß.

Man könnte aber auch das Betriebssystem ins RAM ab $E000 kopieren und diejenigen Bereiche, in denen nicht benötigte Routinen stehen (zum Beispiel für Kassettenbetrieb) einfach überschreiben. Dies ist dann sinnvoll, wenn nur ein paar Byte im $E000-Bereich gebraucht werden. Außerdem ist eine gute Kenntnis des C 64-ROMs erforderlich.

Nun wollen wir noch besprechen, wie der Speicherplatzbedarf eines Programms niedriggehalten werden kann. Dazu wurde im Laufe des Kurses schon einiges gesagt (Unterprogramme statt Makros verwenden etc.).

Jedes Programm benötigt eine Menge Flags. Meist belegt ein Flag genau 1 Byte, für dessen Inhalt es oft nur zwei mögliche Werte gibt: einen für »JA« und einen für »NEIN«.

Für diese primitive Unterscheidungsform genügt aber auch 1/8 Byte, also ein Bit.

Wenn Sie sich das 64'er Extra in der Ausgabe 10/85 ansehen, werden Sie feststellen, daß fast jedes VIC-Register mehrere Funktionen hat, weil jedem Bit eine eigene Bedeutung zukommt. Würde der VIC hier statt auf Bits auf Bytes zugreifen müssen, wäre er

1. langsamer und
2. würde der Speicherplatzaufwand für die Register sich vervielfachen.

Man sollte also bei Flags jedem Bit eine Bedeutung geben und nur die Bits prüfen:

BIT FLAG

Danach ist das N-Flag gesetzt, falls das 7. Bit im FLAG gesetzt ist, und das V-Flag, falls das 6. Bit gesetzt ist. Die übrigen Flags erhält man über das Z-Flag im Prozessor-Status-Register mit Hilfe des Akkus. Angenommen, man möchte testen, ob Bit 0 im Flag gesetzt ist oder nicht, dann macht das folgendes Programm:

```
LDA #01
BIT Flag
BNE ???        ; (Bit gesetzt)
.
.
.
; (Bit nicht gesetzt)
```

Der Bit-Befehl ANDet den Inhalt des Akkus mit dem Inhalt der Speicherzelle »Flag«. Möchte man Bit 1 testen, so ist der Befehl LDA #01 zu ersetzen durch LDA #02 und so weiter.

Durch Selbstmodifikation können Flags bekanntlich vermieden werden. Aber auch sonst bietet die Selbstmodifikation die Möglichkeit, Speicherplatz zu sparen: die Steuerung einer Sprungtabelle belegt mit Selbstmodifikation weniger Speicher als ohne.

Auch die »Wegwerfmethode« ist sehr vorteilhaft. Programmteile werden einmal abgearbeitet und dann (zum Beispiel durch Nachladen) überschrieben.

Damit hätten wir unseren Kurs abgeschlossen. Ich hoffe, daß er Ihnen etwas Spaß gemacht hat und Sie einige interessante Informationen herausholen konnten. Sie sollten sich jedoch darüber im klaren sein, daß einige der hier vorgestellten Methoden die Lesbarkeit eines Assembler-Listings einschränken können. Also, verzichten Sie, wenn nicht unbedingt notwendig, auf allzu trickreiche Programmierung. Falls Sie noch Fragen oder Probleme haben (vielleicht erst wegen diesem Artikel), dann schreiben Sie doch einfach.

(Florian Müller/tr)