# Von Basic zu Assembler (Teil 3)

**Wozu lassen sich Schleifen in Maschinensprache einsetzen? Anhand von Beispielen zeigen wir Ihnen, wie man den Grafikspeicher löscht und beschreibt oder einen Rahmen um den Bildschirm legt. Auch das Beschreiben eines Bildschirmfensters wird in dieser Folge erwähnt. Diese universell zu gebrauchenden Routinen können zum Aufbau einer Makrobibliothek verwendet werden.**

Vier Anwendungen der Schleifenprogrammierung werden Sie in dieser Folge kennenlernen: Das Löschen des Grafikspeichers, das Beschreiben des Grafik-Farbspeichers, einen Rahmenaufbau um den Bildschirm und schließlich das Beschreiben eines Bildschirmfensters.

Dies ist gleichzeitig auch die erste Reaktion auf Ihre Fragen, die Sie uns zum Thema Assembler gesandt haben. Die am Ende der letzten Folge versprochene Vorstellung der BLITUC-Routine muß noch ein wenig warten. Noch eine technische Vorbemerkung: Alle Programme wurden mit einem Nachfolger des Hypra-Ass — dem Programm Top-Ass — geschrieben. Aus dem Top-Ass wurden aber nur die Optionen verwendet, die auch in Hypra-Ass enthalten sind. Einige Pseudo-Opcodes heißen etwas anders, die Bedeutung ist aber leicht zu erkennen. In den Kommentaren finden Sie den jeweiligen Befehl auch in der Hypra-Ass-Syntax.

In der letzten Folge hatten wir uns eine spezielle Form der Doppelschleife angesehen, die es möglich machte, auch von ganzen Seiten (Pages) abweichende Speicherbereiche zu bearbeiten. Als Beispiel hatten wir den Bildschirminhalt invertiert. Diese Doppelschleife soll in verallgemeinerter Form diesmal Verwendung finden. Bild 1 zeigt Ihnen ein Flußdiagramm dieser allgemeinen 16-Bit-Schleife:

Schon in dem Beispiel zur Invertierung hatten wir die Startadresse als Vektor in zwei Zeropage-Speicherstellen geschrieben. Nun wird auch die Endadresse als Vektor $FC/FD gespeichert. So braucht nur im Initialisierungsteil von Aufgabe zu Aufgabe eine Änderung vorgenommen zu werden. Noch allgemeiner kann die Doppelschleife gestaltet werden durch eine Änderung des Jobteils. Handelt es sich beispielsweise um die Aufgabe, bestimmte Speicherbereiche zu beschreiben, dann kann auch der einzuschreibende Wert in eine Zeropage-Speicherstelle gepackt werden (hier in $FE). Will man allerdings auch die Art des Jobs offenhalten, dann verwendet man lediglich Sprünge in Unterprogramme. Der Jobteil heißt dann nur noch:

JSR JOB

Unsere Aufgabe ist es dann, an der Stelle JOB jeweils das gebrauchte Unterprogramm bereitzuhalten. Allerdings sollten solche Schleifenformen nicht oft benutzt werden, denn die Sprünge ins Unterprogramm verbrauchen relativ viel Rechenzeit.

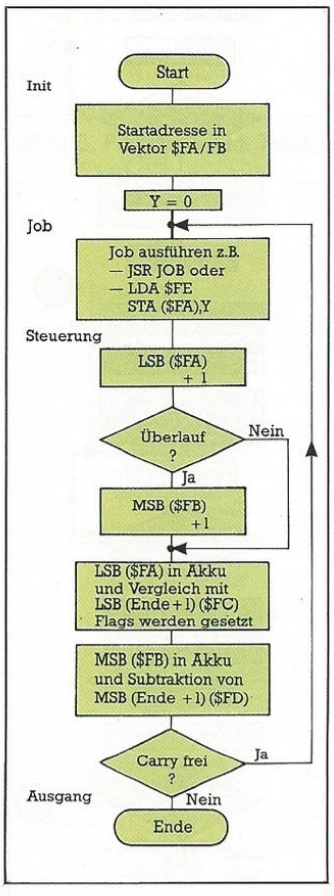



**Bild 1. Flußdiagramm einer Doppelschleife für beliebige Laufzahlen**

## Grafik-Farbspeicher belegen

Zwei Fragen treten häufig auf, die das Löschen oder Neubeschreiben der Grafikspeicher betreffen. Beide sind mit einund derselben Doppelschleife lösbar. Sehen wir uns zunächst einmal die Sache mit dem Grafik-Farbspeicher an. Im allgemeinen verwendet ein C 64-Grafik-Programmierer eine Bit-Map, die bei 8192 startet und ein Farb-RAM, das anstelle des normalen Bildschirmes (also ab 1024 = $400) zu finden ist. Der Benutzer des C 128 hat die Bit-Map am gleichen Ort, aber dafür ein extra Grafik-Farb-RAM ab $1C00. Zwar hat das Basic 7.0 des C 128 allerlei nette Grafik-Befehle anzubieten: Wenn aber gewünscht wird, eine schon auf dem Bildschirm sichtbare Zeichnung mit anderen Farben zu zeigen, stehen beide (also C 64- und C 128-Benutzer) vor demselben Problem. Eine globale Farbänderung ist beim C 128 nämlich nur bei gleichzeitigem Löschen der Bit-Map möglich! Wie also ist die Aufgabe lösbar?

In beiden Fällen ist in 1000 Speicherstellen ein bestimmter Wert einzuschreiben: Beim C 64 von $400 bis $7E8, beim C 128 von $01C00 bis $01FE8. Der Farbcode, der einzutragen ist, hat in beiden Fällen denselben Aufbau: Das untere Nibble (also die Bits 0 bis 3) enthält den Code der Hintergrund-, das obere Nibble (Bits 4 bis 7) den der Zeichenfarbe. C 128-Benutzer müssen vom Farbcode jeweils noch eine 1 abziehen. Sei ZF die Zeichen- und HF die Hintergrundfarbe, dann folgt für den einzuschreibenden Code F:

F = 16 * ZF + HF

Als Listing 1 finden Sie eine mögliche Lösung des Problems. Hier wurde in der Initialisierung (Zeilen 110 bis 270) auch die Belegung der Zeropage-Adressen mit dem Startwert ($FA/FB), dem Endwert ($FC/FD) und dem Farbcode ($FE) vorgenommen:

Zum Listing selbst muß sicher nichts mehr gesagt werden: Es ist ausführlich kommentiert und entspricht genau der oben im Flußdiagramm gezeigten Doppelschleife. Zum Starten dieses Maschinenprogrammes: Wenn Sie es einfach durch

```
SYS 49152
(C 128: BANK 0:SYS49152)
```

mit auf dem Bildschirm vorhandener Abbildung ablaufen lassen, wird der Farbcode $F0 eingetragen, also hellgrau auf schwarzen Hintergrund. Falls Sie im Textmodus sind, entspricht das dem Zeichen mit dem Code $F0 (dezimal 240, ein inverses Grafik-Zeichen), das nun den gesamten Bildschirm ziert. Möchten Sie eine andere Farbenkombination erzielen, dann haben Sie mehrere Möglichkeiten. Entweder lassen Sie einfach die Zeilen 230 und 240 wegfallen und POKEn vor dem Routinenaufruf Ihren Wert F in die Speicherstelle $FE oder

```
ready.
10  -.list 1,4,7                ;in hypra-ass: .li 1,4,7
20  -.base $c000                 ;in hypra-ass: .ba $c000
30  -;***********************************************
40  -; 16-bitschleife anwendung : bitmap-loeschen
50  -;***********************************************
60  -;
70  -.define start     = $2000 ;in hypra-ass: .eq start = $2000
80  -.define ende      = $3f3f ;   -"-         .eq ende  = $3f3f
90  -.define wert      = $00   ;   -"-         .eq wert  = $00
100 -;
110 -;---------- initialisierung -----------
120 -;
130 -           lda #<(start)   ;lsb startadresse
140 -           ldy #>(start)   ;msb startadresse
150 -           sta $fa         ;in vektor $fa/fb schreiben
160 -           sty $fb
170 -;
180 -           lda #<(ende)    ;lsb endadresse+1
190 -           ldy #>(ende)    ;msb endadresse
200 -           sta $fc         ;in vektor $fc/fd schreiben
210 -           sty $fd
220 -;
230 -           lda #wert       ;einzuschreibenden wert
240 -           sta $fe         ;nach $fe schreiben
250 -;
260 -           ldy #$00        ;index auf null stellen
270 -;
280 -;---------- job ausfuehren -------------
290 -;
300 -label      lda $fe         ;wert laden
310 -           sta ($fa),y     ;und eintragen
320 -;
330 -;--------- steuerteil -------------------
340 -;
350 -           inc $fa         ;lsb start nun als zaehler erhoehen
360 -           bne marke       ;falls kein ueberlauf weiter
370 -           inc $fb         ;sonst msb ebenfalls erhoehen
380 -marke      lda $fa         ;vergleich des lsb
390 -           cmp $fc         ;mit lsb der endadresse (flaggen setzen)
400 -           lda $fb         ;vom msb des zaehlers
410 -           sbc $fd         ;wird das msb der endadresse subtrahiert
420 -           bcc label       ;zurueck zum job wenn zaehler < endadresse
430 -;
440 -;-------- ausgang ----------------------
450 -;
460 -           brk             ;sonst programmende
470 -;
480 -.symbols u,1,4,7           ;in hypra-ass: .sy 1,4,7
```

**Listing 1. Ein Programm zum Beschreiben des Grafik-Farb-RAM**

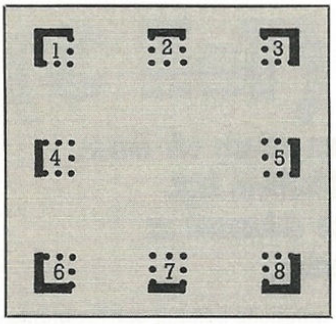



**Bild 2. Zum Rahmenproblem: Welche Zeichen werden benötigt?**

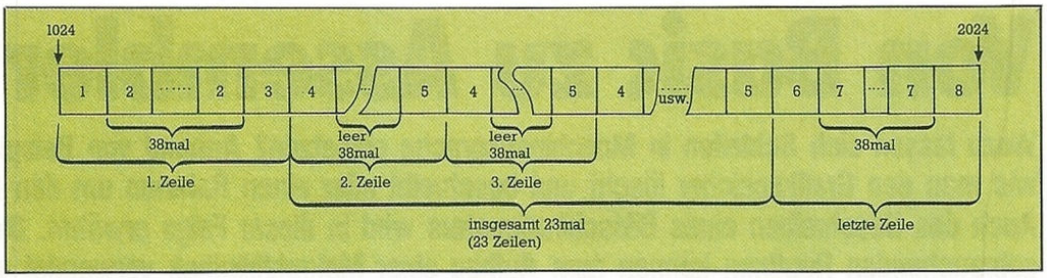



**Bild 3. Zur Verdeutlichung von Bild 2 hier die grafische Darstellung des Rahmenproblems**

aber Sie verändern das Maschinenprogramm vor dem Aufruf, indem Sie an die Stelle in Zeile 230, an der der WERT steht, Ihr F POKEn. Das ist (vorausgesetzt, Sie belassen den Start bei $C000) die Speicherzelle $C011 (das ist dezimal 49169). Ein Aufruf könnte dann beispielsweise so aussehen:

```
10 INPUT"ZF,HF =";ZF,HF
20 F = 16*ZF + HF
30 POKE 49169,F
40 SYS 49152
```

Auf ähnliche Weise können natürlich auch die Start- und/oder Endadressen variiert werden, so daß beispielsweise nur der halbe Farbspeicher neu belegt wird. C128-Benutzer müssen zum Ändern des Grafik-Farb-RAM die Werte in den Zeilen 70 und 80 auf die oben genannten Adressen einstellen.

## Bit-Map löschen

Vom eben gezeigten Beispiel zum Löschen einer Bit-Map ist es nur ein kleiner Schritt. Lediglich der Ort einer Bit-Map ist ein anderer und ihr Umfang. Im allgemeinen startet der Grafik-Speicher bei 8192 ($2000), hat eine Ausdehnung von 8000 Bytes (nämlich 200*320/8) und endet bei $3F3F (dezimal 16191). Der einzutragende Wert ist Null. Listing 1 unterscheidet sich vom Listing 2 somit nur durch die Zeilen 70 bis 90.

Daran können Sie ersehen, wie vielseitig unsere Doppelschleife ist. C 128-Benutzer werden nun meinen, daß sie das Listing 2 und — falls ihnen niemals das Problem der Farbänderung eines fertigen Bildes unterkommt — auch das Listing 1 entbehren könnten. Sollten sie aber jemals in die Lage kommen, eine andere Bit-Map und einen anderen Grafik-Farbspeicher als die softwaremäßig voreingestellten bedienen zu wollen, dann sind sie in der gleichen Lage wie ein C64-Benutzer. Alle Grafik-Kommandos zielen nur auf die oben erwähnten Standard-Grafik-Speicherbereiche. Jede andere Bit-Map und jedes andere Farb-RAM muß mit selbst gebastelten Befehlssequenzen behandelt werden.

Noch eine kleine Bemerkung am Rande: Falls es Ihnen in den Sinn kommen sollte, das zum Listing 2 gehörende Maschinenprogramm einfach mal zu starten, dann bedenken Sie, daß durch Hypra-Ass im fraglichen — zu löschenden — Speicherbereich Quelltexte und Label abgelegt sind. Speichern Sie sie also vor dem Start ab! Noch schlimmer ergeht es C 128-Benutzern, die den Top-Ass verwenden: Das Löschen des Grafik-Bereiches reißt ein tiefes Loch ins Top-Ass-Programm. Top-Ass und normale Grafik können nicht gleichzeitig betrieben werden.

## Bildschirm umrahmen

Viele Leser wollten wissen, wie man in Assembler einen Rahmen um den Textbildschirm legen kann. Gehen wir also dieses Problem möglichst allgemeingültig an. Wir brauchen dazu acht verschiedene Grafikzeichen (siehe dazu das Bild 2). Außerdem erkennen wir recht schnell, daß wir bei diesem Problem andere Schleifen benutzen müssen. In Bild 3 ist die Aufgabe grafisch aufgeschlüsselt.

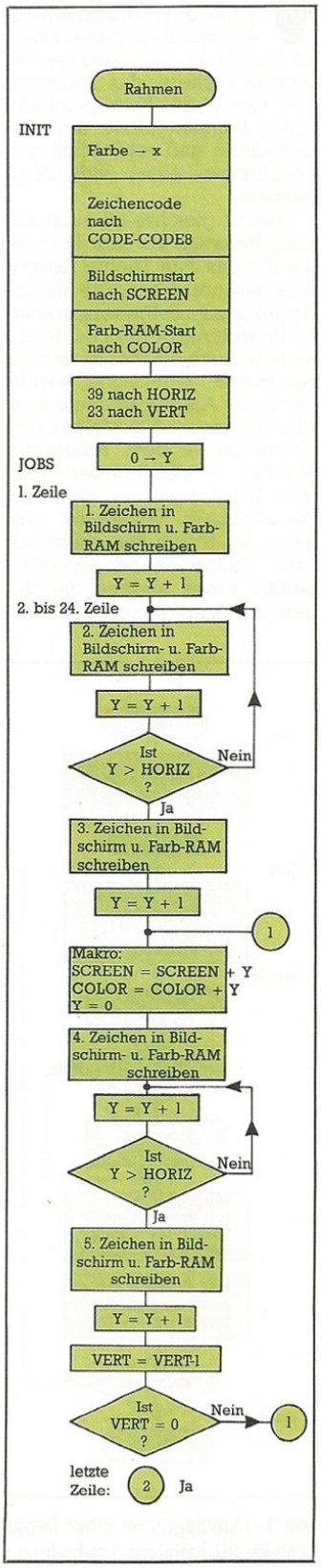
Die Nummern entsprechen jeweils den verschiedenen Zeichen, die in eine Speicherstelle einzuschreiben sind. Es handelt sich um folgende:

| Nr. | Hex. | Dez. | Beschreibung |
|---|---|---|---|
| 1 | 4F | 79 | Winkel links oben |
| 2 | 77 | 119 | Linie oben |
| 3 | 50 | 80 | Winkel rechts oben |
| 4 | 74 | 116 | Linie linksseitig |
| 5 | 6A | 106 | Linie rechtsseitig |
| 6 | 4C | 76 | Winkel links unten |
| 7 | 6F | 111 | Linie unten |
| 8 | 7A | 122 | Winkel rechts unten |

Im Listing 3 sind diese acht Zeichen in der Initialisierungsphase in acht Zeropage-Speicherstellen geschrieben worden ($1B bis $22), die CODE1 bis CODE8 genannt wurden.

Bevor wir dieses Programm besprechen, gebe ich Ihnen in Bild 4 noch das Flußdiagramm dazu an.

Das Programm ist in drei Teilen angeordnet: Die Initialisierung, in der alle Startwerte festgelegt werden. Sie liegen durchweg in Zeropage-Speicherstellen, was durch einfaches Ändern dieses Programmteils das Programm flexibel hält. Wir hätten noch einen Schritt weiter gehen können, indem wir



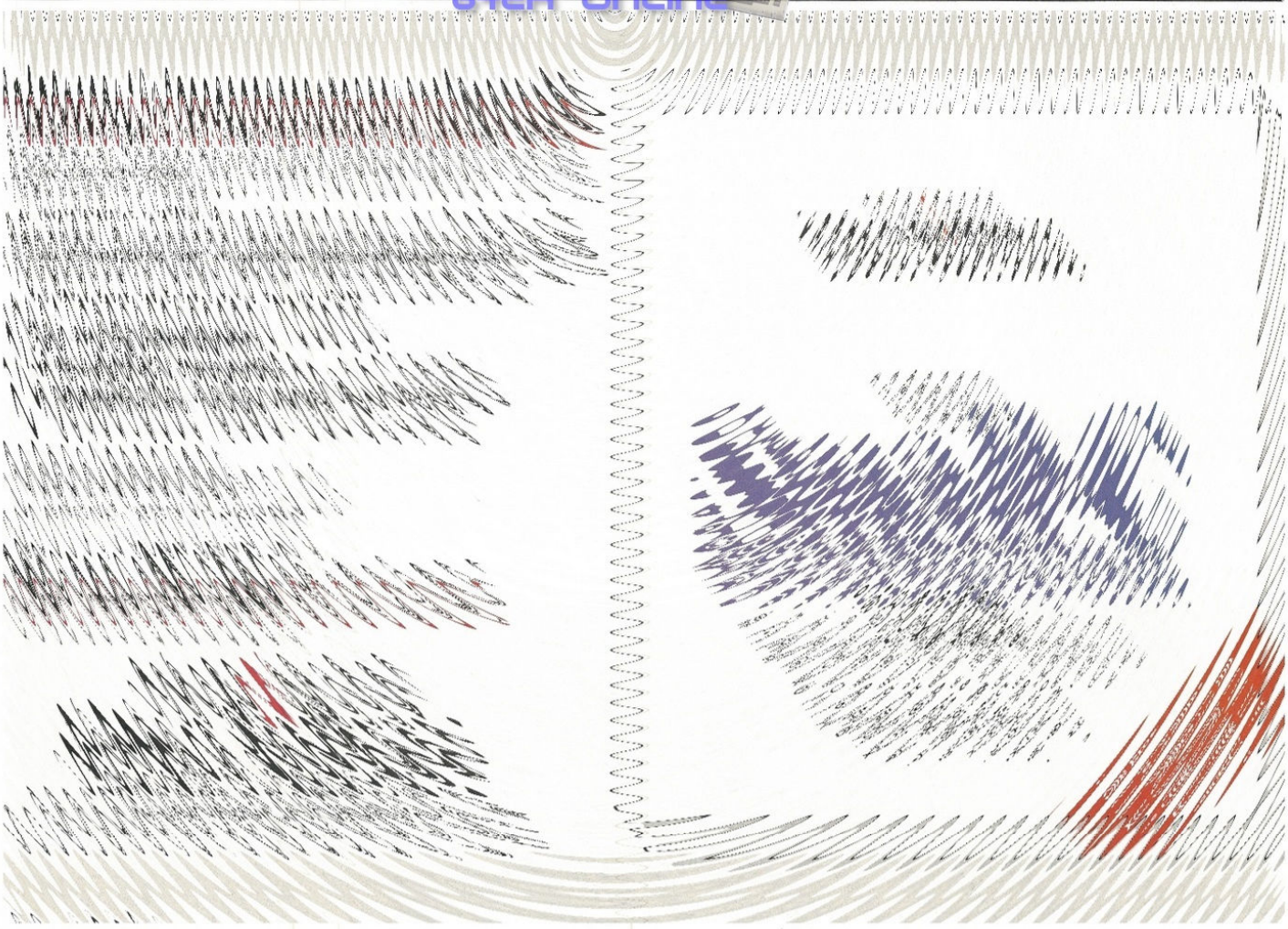

```
10   -.list 1,4,7                ;in hypra-ass: .li 1,4,7
20   -.base $c000                ;in hypra-ass: .ba $c000
30   -;***********************************************
40   -; 16-bitschleife anwendung : screen-speicher
50   -;***********************************************
60   -;
70   -.define start      = $0400 ;in hypra-ass statt .define
80   -.define ende       = $07e8 ;jeweils: .eq  zum beispiel
90   -.define wert       = $f0   ; .eq start = $0400
100  -;
110  -;---------- initialisierung ------------
120  -;
130  -            lda #<(start)  ;lsb startadresse
140  -            ldy #>(start)  ;msb startadresse
150  -            sta $fa        ;in vektor $fa/fb schreiben
160  -            sty $fb
170  -;
180  -            lda #<(ende)   ;lsb endadresse+1
190  -            ldy #>(ende)   ;msb endadresse
200  -            sta $fc        ;in vektor $fc/fd schreiben
210  -            sty $fd
220  -;
230  -            lda #wert      ;einzuschreibenden wert
240  -            sta $fe        ;nach $fe schreiben
250  -;
260  -            ldy #$00       ;index auf null stellen
270  -;
280  -;---------- job ausfuehren -------------
290  -;
300  -label      lda $fe        ;wert laden
310  -            sta ($fa),y    ;und eintragen
320  -;
330  -;--------- steuerteil --------------------
340  -;
350  -            inc $fa        ;lsb start nun als zaehler erhoehen
360  -            bne marke      ;falls kein ueberlauf weiter
370  -            inc $fb        ;sonst msb ebenfalls erhoehen
380  -marke      lda $fa        ;vergleich des lsb
390  -            cmp $fc        ;mit lsb der endadresse (flaggen setzen)
400  -            lda $fb        ;vom msb des zaehlers
410  -            sbc $fd        ;wird das msb der endadresse subtrahiert
420  -            bcc label      ;zurueck zum job wenn zaehler < endadresse
430  -;
440  -;--------- ausgang ------------------------
450  -;
460  -            brk            ;sonst programmende
470  -;
480  -.symbols u,1,4,7           ;in hypra-ass: .sy 1,4,7
```

**Listing 2. Blitzartiges Löschen der Bit-Map**

```
10   -.list 1,4                    ;in hypra-ass: .li 1,4
20   -.base $c000                  ;in hypra-ass: .ba $c000
30   -;**************************************
40   -; verschachtelte schleifen  anwendung
50   -; allgemeiner bildschirmrahmen
60   -;**************************************
70   -;
80   -;lagerplatz fuer die zeichen: (hpyra-ass jeweils: .eq code1 = $1b usw.)
90   -.define code1      = $1b    ;zeichen 0 = 4f
100  -.define code2      = $1c    ;zeichen 7 = 77
110  -.define code3      = $1d    ;zeichen P = 50
120  -.define code4      = $1e    ;zeichen % = 74
130  -.define code5      = $1f    ;zeichen ' = 6a
140  -.define code6      = $20    ;zeichen L = 4c
150  -.define code7      = $21    ;zeichen / = 6f
160  -.define code8      = $22    ;zeichen ; = 7a
170  -;vektoren fuer bildchirm- und farb-ram:
180  -.define screen     = $fb    ;bildschirmstart
190  -.define color      = $fd    ;farbramstart
200  -;zaehler:
210  -.define horiz      = $23    ;zaehler fuer horizontale
220  -.define vert       = $24    ;zaehler fuer vertikale
230  -;der farbcode wird nur im x-register gespeichert
240  -;
250  -;definition eines makro: aktuell
260  -;in hypra-ass stattdessen: .ma aktuell(screen,color)
270  -.macro aktuell(screen,color)
280  -            clc
290  -            tya
300  -            adc screen
310  -            sta screen
320  -            lda screen+1
330  -            adc #$00
340  -            sta screen+1
350  -            clc
360  -            tya
370  -            adc color
380  -            sta color
390  -            lda color+1
400  -            adc #$00
410  -            sta color+1
420  -            ldy #$00
430  -.macend                       ;in hypra-ass: .rt
440  -;
450  -;nun gehts los:
460  -;
470  -;-------------- initialisierung ------------
480  -;
490  -            ldx #$02          ;farbcode ins x-register
500  -            lda #$4f          ;alle zeichen in die speicher packen
510  -            sta code1
520  -            lda #$77
530  -            sta code2
540  -            lda #$50
550  -            sta code3
560  -            lda #$74
570  -            sta code4
580  -            lda #$6a
590  -            sta code5
600  -            lda #$4c
610  -            sta code6
620  -            lda #$6f
630  -            sta code7
640  -            lda #$7a
650  -            asl bild          ;=  *16
660  -            rol bild+1        ;zur sicherheit carry in msb rotieren
670  -            asl bild          ;=  *32 (zweiter summand)
680  -            rol bild+1        ;nochmal carry ins msb rotieren
690  -            lda zwsp          ;addition beider summanden (carry ist frei)
700  -            adc bild          ;lsb von 32*z1
710  -            sta bild          ;lsb von 40*z1
720  -            lda bild+1        ;msb von 32*z1
730  -            adc #$00          ;eventuell carry addieren
740  -            sta bild+1        ;im vektor bild steht nun 40*(z1-1)
750  -;
760  -            clc               ;addition von s1
770  -            lda bild          ;lsb
780  -            adc spalte        ;das ist ja s1
790  -            sta bild
800  -            lda bild+1        ;msb
810  -            adc #$00          ;eventuell carry addieren
820  -            sta bild+1        ;im vektor bild steht nun 40*(z1-1)+s1
830  -;
840  -            clc               ;farbramadresse berechnen
850  -            lda #<(colram)    ;lsb
860  -            adc bild
870  -            sta color         ;vektor aktuelle farbramadresse lsb
880  -            lda #>(colram)    ;msb
890  -            adc bild+1
900  -            sta color+1       ;vektor msb
910  -;
920  -            clc               ;aktuelle bildschirmadresse berechnen
930  -            lda #<(screen)    ;lsb
940  -            adc bild
950  -            sta bild          ;vektor bildschirm lsb
960  -            lda #>(screen)    ;msb
970  -            adc bild+1
980  -            sta bild+1        ;bildschirmvektor msb
990  -;
1000 -marke5      ldx #farbe        ;durch poke veraenderbar
1010 -;
1020 -;******* hier beginnen die verschachtelten schleifen *******
1030 -label1      ldy #$00          ;spaltenzaehler zuruecksetzen
1040 -label2      lda #wert         ;durch poke veraenderbar
1050 -            sta (bild),y      ;auf bildschirm schreiben
1060 -            txa               ;farbwert umladen
1070 -            sta (color),y     ;und in farbram schreiben
1080 -            iny
1090 -            cpy spdiff        ;zeile fertig ?
1100 -            bcc label2        ;noch nicht, dann weitermachen
1110 -;ansonsten werden zu den vektoren bild und color 40 addiert:
1120 -            clc               ;bild aktualisieren
1130 -            lda bild          ;lsb
1140 -            adc #$28          ;das ist dezimal 40
1150 -            sta bild
1160 -            lda bild+1        ;msb
1170 -            adc #$00          ;carry addieren
1180 -            sta bild+1        ;vektor bild ist aktualisiert
1190 -;
1200 -            clc               ;color aktualisieren
1210 -            lda color         ;lsb
1220 -            adc #$28          ;wieder 40 dazuzaehlen
1230 -            sta color
1240 -            lda color+1       ;msb
1250 -            adc #$00          ;addieren des carry
1260 -            sta color+1       ;vektor color ist aktualisiert
1270 -;
1280 -            dec zdiff         ;zeilenzaehler herunterzaehlen
1290 -            bne label1        ;wenn noch > 0, naechste zeile schreiben
1300 -;
1310 -;im anderen fall ist die aufgabe erledigt:
1320 -            rts               ;zurueck zum aufrufenden programm oder basic
1330 -;
1340 -.symbols u,1,4,7             ;in hypra-ass: .sy 1,4,7
```

**Listing 5.**
**Das Assembler-Programm »Teilbereich«**





Zeile 80 statt des DEC ("3FF") einfach 1023 ein.

Natürlich funktioniert das so einwandfrei, aber es dauert! In Assembler geht das Ausfüllen auch dann noch blitzartig, wenn wir als Rechteck den gesamten Bildschirm vorgeben. Das einzige, was Kopfzerbrechen bereiten kann, ist die Berechnung der Startadresse. Aber auch dazu gibt es einen schnellen Weg, wie Sie gleich noch sehen werden. Bild 7 gibt Ihnen als Anhaltspunkt ein Flußdiagramm zum Assemblerprogramm TEILBEREICH.

Listing 5 schließlich enthält unser gleich zu besprechendes Programm »Teilbereich«.

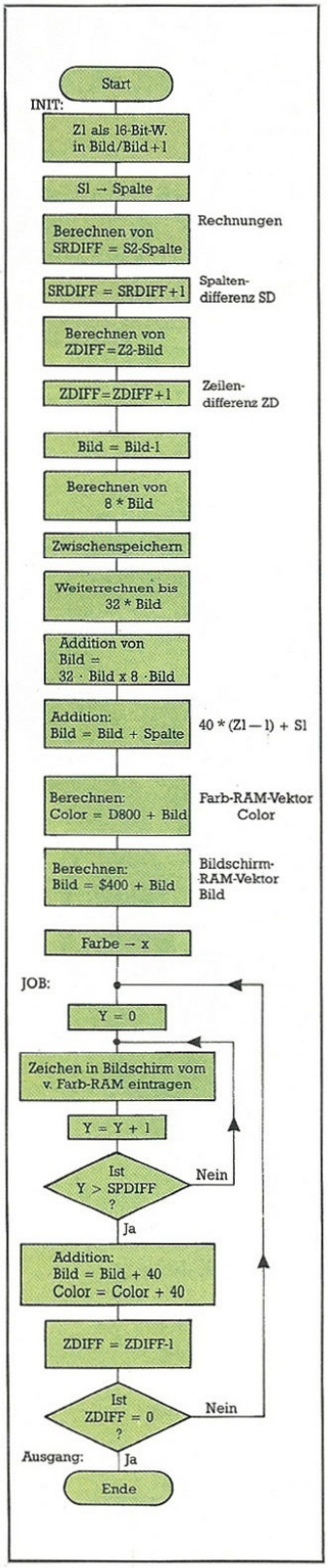



**Bild 7. Flußdiagramm zum Programm »Teilbereich«**

Im ersten Teil des Listings sehen Sie wieder die Definitionen, die der Assembler beim Assemblieren statt der Merktexte einsetzt. Diesmal sind von Zeile 130 bis 180 Werte vorgegeben (purpurfarbene Zeichen A werden in das Rechteck geschrieben, das von Spalte 4/Zeile 3 bis Spalte 15/ Zeile 13 reicht), von 220 bis 270 Zeropage-Speicherstellen, die der Speicherung der Spaltendifferenz (SPDIFF), Zeilendifferenz (ZDIFF), der Spalte S1 (SPALTE) und eines Zwischenwertes der Rechnung (ZWSP) dienen. Außerdem sind zwei Vektoren vorgesehen: BILD (für die laufende Bildschirmspeicherposition) und COLOR (dasselbe für das Farb-RAM). Die Zeilen 310 und 320 enthalten die Festlegung der Basisadressen des Bildschirmspeichers und des Farb-RAMs (jeweils -1).

Der erste Teil des eigentlichen Programmes dient der Initialisierung und der Berechnung der Steuergrößen (im Basic-Programm waren das SD, ZD und S). Zunächst speichern wir Z1 als 16-Bit-Wert im Vektor BILD. Dazu muß das MSB von BILD (also BILD+1) noch auf Null gesetzt werden, denn Z1 ist ja nur ein 8-Bit-Wert. Der Wert S1 wird danach in SPALTE geschrieben (Zeilen 430 und 440). Nun fängt die Rechnerei an. Wir bilden die Spalten- und die Zeilendifferenzen jeweils durch einfache 8-Bit-Subtraktionen. Zur Erinnerung: Vor einer normalen Subtraktion muß immer das Carry-Bit gesetzt werden, denn es dient gewissermaßen als »Stelle, die geborgt werden kann«. Die Wirklichkeit ist etwas komplexer, aber das können Sie — falls es Sie interessiert — im Assemblerkurs oder einschlägigen Lehrbüchern nachlesen. Nach vollendeter Subtraktion werden SPDIFF und ZDIFF jeweils noch durch den INC-Befehl um 1 erhöht. Das hat seinen Grund in den Schleifenabbruchbedingungen im 2. Teil unseres Programmes. Soweit war die Sache noch recht einfach. Nun kommt aber die Berechnung des Startpunktes im Bildschirm- und im Farb-RAM. Das heißt wir haben den Ausdruck

40*(Z1-1)+S1

zu berechnen. Wir bedienen uns in der Hauptsache der Assembler-Befehle ASL und ROL dazu, die — weil wir sie recht selten erleben — hier nochmal kurz erläutert werden sollen. Die Bilder 8 und 9 illustrieren die Erklärungen.

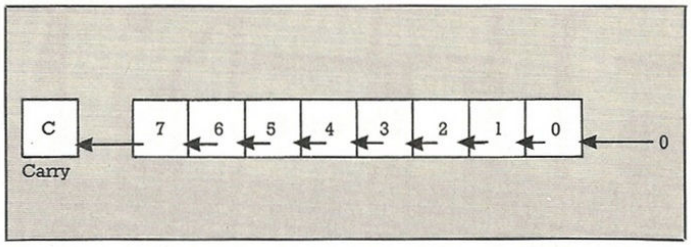



**Bild 8. Die grafische Darstellung der Wirkung eines ASL-Befehls**

ASL (arithmetic shift left = arithmetisches Linksverschieben) schiebt jedes Bit des angesprochenen Bytes um eine Position nach links. An die Stelle des Bit 0 tritt eine Null und das Bit 7 landet im Carry. Was das bedeuten kann, sehen wir uns an einer Zahl im Dezimalsystem mal an:

240 unsere Zahl
2400 nun ist jede Stelle einmal nach links verschoben

Sie sehen, daß das einer Multiplikation um den Faktor 10 entspricht. Im binären Zahlensystem, das unser Computer verwendet, ist die Zahlenbasis nicht mehr 10, sondern 2. Deshalb führt hier jede Linksverschiebung zu einer Multiplikation mit 2. Sehen wir uns das am konkreten Beispiel an: Der höchste Wert für Z1 in unserem Programm ist 25 (es gibt nur 25 Zeilen). Wenn wir von Z1-1 ausgehen ist das 24:

0001 1000 unsere Zahl 24
1.ASL: 0011 0000 das ist 2*24
2.ASL: 0110 0000 das ist 4*24
3.ASL: 1100 0000 das ist 8*24

Vorsicht! Wenn wir nun nochmal die ASL-Operation anwenden, schieben wir das Bit 7 in das Carry-Bit. Das Ergebnis ist für eine 8-Bit-Zahl zu groß geworden, eine 16-Bit-Zahl ist vonnöten. Wir müßten dann die 1 aus dem Carry-Bit als Bit 0 des MSB verwenden können. Das wird mit ROL (rotate left = nach links rotieren) möglich. Auch verschiebt der Befehl jedes Bit um eine Stelle nach links. Als neues Bit 0 aber tritt der Inhalt des Carry-Bit ein. Parallel dazu wandert das Bit 7 ins Carry-Bit. Der weitere Weg in unserem Beispiel muß nun also lauten:

4.ASL: 1 1000 0000 und dann
1.ROL: 0000 0001 1000 0000
MSB LSB

Das ist nun 16*24. Durch ASL wurde Bit 7 ins Carry geschoben (deshalb steht es oben links außen), durch ROL (bezogen auf die Speicherstelle, die als MSB eingesetzt wird) wandert es dann ins MSB als Bit 0. Die dazugehörige Befehlssequenz lautet (siehe beispielsweise Zeilen 650/660 des Listing 5):

ASL BILD
ROL BILD+1

Vermutlich werden Sie sich nun fragen, was das alles mit unserem Problem zu tun hat, 40*(Z1-1)+S1 zu berechnen. Sehr viel! Rechnen Sie es mal nach. Es gilt nämlich:

40*(Z1-1) = 8*(Z1-1) + 32*(Z1-1)

Sowohl 8 als auch 32 sind aber Potenzen von 2, das bedeutet, daß sie durch mehrmaliges Multiplizieren der 2 mit sich selbst herauskommen: 2^3 = 8 und 2^5 = 32. Es ist damit möglich, durch mehrmaliges Anwenden der ASL-Operation auf (Z1-1) beide Summanden zu erzeugen. Die Addition ergibt dann unseren gewünschten Wert 40*(Z1-1). Wir müssen nur noch S1 hinzurechnen. Wie wir vorhin noch sehen konnten, dürfen wir die drei ersten ASL-Operationen noch ohne Rücksicht auf Verluste durchführen. Danach allerdings muß jedem ASL das dazugehörige ROL folgen, denn das Ergebnis kann eine 16-Bit-Zahl sein.

In unserem Programm beginnen wir diese Berechnung in Zeile 580. Dort wird zunächst sicherheitshalber das Carry-Bit-

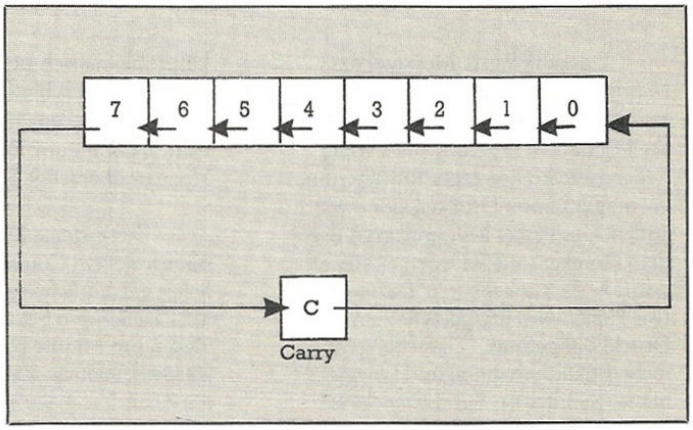



**Bild 9. Die grafische Darstellung der Wirkung des ROL-Kommandos**

```
10 REM ******** AUFRUFPROGRAMM FUER TEILBILDSCHIRM
 **********
20 REM IN REM-BEMERKUNGEN JEWEILS WERTE FUER DEN C
128
30 REM VORAUSSETZUNG, DASS BEIM C64 AB $C000
40 REM                      BEIM C128 AB $1400 DAS
PROGRAMM LIEGT.
50 REM
60 REM ------- EINGABE DER PARAMETER ---------
70 PRINT CHR$(147):INPUT"LINKE OBERE ECKE Z1,S1";Z
1,S1
80 INPUT"RECHTE UNTERE ECKE Z2,S2";Z2,S2
90 INPUT"FARBE,WERT";F,W
100 REM ------- EINPOKEN DER WERTE ------------
110 POKE 49157,Z1:REM POKE 5125,Z1
120 POKE 49161,S1:REM POKE 5129,S1
130 POKE 49166,S2:REM POKE 5134,S2
140 POKE 49175,Z2:REM POKE 5143,Z2
150 POKE 49255,F:REM POKE 5223,F
160 POKE 49259,W:REM POKE 5227,W
170 REM ------ AUFRUF DER ROUTINE ------------
180 SYS49152
190 END
```

**Listing 6. Ein Basic-Programm, das alle POKE-Befehle ausführt und unsere Routine ansteuert**

gelöscht für das erste ROL-Kommando. Durch DEC BILD erzeugen wir den Wert Z1-1 (nach BILD hatten wir ja Z1 geschrieben, BILD+1 wurde reserviert für das MSB und eine Null eingetragen). Dreimaliges ASL BILD erzeugt 8*(Z1-1). Sie erinnern sich: Erst vom vierten ASL an mußte beim größten erlaubten Z1-Wert auch ROL eingesetzt werden. In den Zeilen 640/650 packen wir diesen Summanden in den Zwischenspeicher. Danach sind noch zwei ASL mit anschließenden ROL-Kommandos nötig, um in BILD/BILD+1 den Wert 32*(Z1-1) zu erzeugen. Von Zeile 690 bis 740 addieren wir beide Summanden und erhalten so 40*(Z1-1). Normalerweise muß vor jeder Addition mit ADC (denn das heißt ja »add with carry«, also addiere mit dem Carry-Bit, sogar addiere auch das Carry-Bit) das Carry-Bit durch CLC freigemacht werden. Hier können wir uns das aber ersparen, weil wir wissen, daß auch der größte Z1-Wert bei der letzten ROL-Operation nur eine Null ins Carry-Bit schieben konnte. Bei der folgenden Addition von S1 (Zeilen 760 bis 820) entsteht schließlich unser gewünschter Startwert — aber noch ohne Berücksichtigung der Anfangsadressen von Bildschirm- und Farb-RAM. Das kommt nun: Von Zeile 840 bis 900 entsteht im Vektor COLOR/COLOR+1 der Zeiger auf die erste Farb-RAM-Stelle, von Zeile 920 bis 980 im Vektor BILD/BILD+1 schließlich der auf die erste Bildschirmspeicherstelle. Zum Abschluß dieser Programmphase schieben wir noch den Farbcode ins X-Register.

Nun kommt die Doppelschleife, die den Bildschirmausschnitt beschreibt. Das Y-Register dient als Zähler jeweils einer Zeile. Der Zeichencode gelangt — ebenso wie der Farbcode — nach dem gleichen Rezept in die notwendigen Speicherstellen, das wir auch schon in den vorangegangenen Programmbeispielen gezeigt haben (das war die Sequenz, die wir dort auch als Makro hätten schreiben können:

```
BANK15:SYS DEC("1400").
```

Sehr flexibel wird »Teilbildschirm« aber erst durch einige POKE-Kommandos ins Programm hinein. Folgende Speicherstellen werden dann angesteuert:

| Parameter | Ort im Programm | POKE-Adressen C 64 | C 128 |
|---|---|---|---|
| Z1 | MARKE1+1 | 49157 | 5125 |
| S1 | MARKE2+1 | 49161 | 5129 |
| S2 | MARKE3+1 | 49166 | 5134 |
| Z2 | MARKE4+1 | 49175 | 5143 |
| Farbe | MARKE5+1 | 49255 | 5223 |
| Wert | LABEL2+1 | 49259 | 5227 |

Hier in den Zeilen 1050 bis 1080). Jeweils am Ende der kleinen Schleife (in 1090 und 1100) wird der aktuelle Y-Zählerstand verglichen mit der Spaltendifferenz (SPDIFF) und — falls die Zeile noch nicht fertig ist — der nächste Durchlauf eingeleitet. Andernfalls addieren wir sowohl zum Vektor BILD/BILD+1 als auch zu COLOR/COLOR+1 den Wert 40, rücken also eine Zeile runter. In 1280 dient ein DEC ZDIFF dazu, die Anzahl bearbeiteter Zeilen abzustreichen, bis dabei eine Null auftritt. Ist das noch nicht der Fall, wird zurückverzweigt nach LABEL1, wo wir den Y-Zähler neu initialisieren.

Der letzte Teil unseres Programmes besteht lediglich wieder aus einem lapidaren RTS.

Wenn Sie unser Programm nach dem Assemblieren einfach mittels SYS 49152 starten, sind automatisch die Beispielwerte eingesetzt. C 128-Benutzern sei wieder geraten, das Programm nach $1400 zu legen, damit auch der Farbspeicher belegt werden kann. Hier erfolgt der Start dann durch

Vorausgesetzt wird bei diesen Angaben, daß im C 64 ab $C000 und im C 128 ab $01400 unser Programm zu finden ist. Listing 6 zeigt Ihnen, wie ein Basic-Programm aussehen kann, das alle POKE-Befehle ausführt und unsere Routine ansteuert. Falls geplant ist, Benutzer an dieses Programm zu lassen, die es nicht kennen, sollten Sie auch noch eine Überprüfung der Eingabewerte einbauen, andernfalls könnte beispielsweise Unsinn herauskommen, wenn S2 kleiner als S1 angegeben wird oder mal eine Zeilennummer, die größer als 25 ist. Die erlaubten Werte sind jeweils Spaltenwerte zwischen 1 und 40, Zeilen zwischen 1 und 25.

Wie man Schleifen in Assembler anwenden kann und auf wie verschiedene Arten das möglich ist, haben Ihnen diese vier Beispiele sicher zeigen können. Aber mit Schleifen ist noch viel mehr machbar.

Damit werden wir uns in den nächsten Folgen befassen.

(Heimo Ponnath/dm)

Fortsetzung von Seite 132

daß zwischen Repeat und Until mehrere Anweisungen stehen dürfen (in der While-Schleife muß man eine Verbundanweisung verwenden), da diese Schlüsselwörter die gleiche Wirkung wie Begin und End haben.

Listing 5 zeigt die Anwendung der Repeat-Anweisung. Die Zahlen 1 bis 10 werden innerhalb der Schleife addiert. Zur Verdeutlichung nochmals die einzelnen Schritte:

1. Der Schleifenkörper, das sind die Anweisungen zwischen Repeat und Until, wird ausgeführt.
2. Der logische Ausdruck wird berechnet.
3. Hat der Ausdruck den Wert »true«, ist das Ende der Schleife erreicht. Die nächste Anweisung wird ausgeführt.

Hat der Ausdruck den Wert »false«, so wird die Schleife nochmals durchlaufen.

4. Die Schritte 1 bis 3 werden so oft ausgeführt, bis der logische Ausdruck den Wert »true« annimmt.

In Bild 3 werden diese Schritte nochmals grafisch dargestellt. Wie bei der Verwendung der While-Anweisung muß der Programmierer auch bei der Repeat-Schleife darauf achten, daß er den Wert des Schleifenindexes innerhalb der Schleife nicht verändert. Die Unterschiede zwischen While und Repeat haben wir in Tabelle 1 nochmals zusammengefaßt.

Damit sind alle Anweisungen mit Ausnahme der With-Anweisung vorgestellt. Diese Anweisung werden wir zusammen mit dem Datentyp Record in der übernächsten Folge unseres Kurses besprechen. In der nächsten Folge beschäftigen wir uns mit Ausdrücken und Standardfunktionen und zeigen, wie man sich in Pascal eigene Dateitypen definieren kann. Und außerdem werden die Ausschnitts- und Aufzählungstypen behandelt.

In den weiteren Folgen dieses Kurses stellen wir vor: die Datentypen Array und Record, dann Set (Menge) und Pointer (Zeiger), Funktionen und Prozeduren und schließlich die Dateiverwaltung in Pascal.

(Anton Gruber/
Silvia Gutschmidt/cg)

## Die Sache mit dem Semikolon:

Ein Semikolon trennt zwei Anweisungen voneinander. »Begin« ist keine Anweisung. Aber hinter dem »end« muß ein Semikolon stehen! Vor einem »end« kann die letzte Anweisung mit »;« abgeschlossen werden, muß jedoch nicht. Direkt vor dem »end« muß also kein Semikolon stehen. Vor einem »else« schließlich darf auf keinen Fall ein »;« stehen. Hier würde dieses Trennzeichen einen Syntaxfehler erzeugen.