Der gläserne VC 20

Teil 4

Nachdem wir in der letzten Folge Funktion und Sinn verschiedener Vektoren erläutert haben, wenden wir uns heute einem nicht minder interessanten Kapitel, nämlich den verschiedenen hochauflösenden Grafikmodi zu.

Dieser Teil unserer Serie soll schrittweise zur Gestaltung des Bildschirmes in hochauflösender beziehungsweise vielfarbiger Grafik hinführen. Dazu ist eine genaue Kenntnis der VIC-Register und des Speicheraufbaus notwendig. Wer sich nochmals eine Übersicht über die Aufteilung des Speichers verschaffen will, der kann jetzt in Teil 1 und 3 dieser Serie nachschlagen.

Der VC 20-Zeichensatz

Wenn man den Computer einschaltet, so stehen »serienmäßig« zwei Zeichensätze zur Verfügung (Großschreibung mit Grafikzeichen oder Groß- und Kleinschreibung). Die Umschaltung zwischen beiden erfolgt, wie bekannt, unter anderem über die Tastenkombination CBM + SHIFT.

Was geschieht bei dieser Umschaltung, und warum lassen sich beide Zeichensätze (normalerweise) nicht gleichzeitig darstellen? Um diese Fragen zu klären, betrachten wir im folgenden das Innenleben des VIC (Video Interface Chip).

Dieser Baustein kümmert sich um alles, was mit der Informationsweitergabe an das angeschlossene Fernsehgerät zu tun hat, also Bild, Farbe und Ton. Da er — wie bereits das letzte Mal kurz angedeutet — eigenständig, das heißt unabhängig von der CPU arbeitet, muß der VIC selbständig auf Informationen zurückgreifen können. Diese sind zum einen in den internen Registern, zum anderen in den normalen Speicherstellen enthalten. Die Register speichern Parameter für die Bildschirmgröße, die Tongeneratoren, die Lage des Bildschirm-, Farb- und Zeichenspeichers. Ferner können aus ihnen die Zustandswerte des Paddels und des Lichtgriffels ausgelesen werden (Tabelle 1).

Register	Registerfunktion	Normalwert
0. 36864	ABBB BBBB	5
1. 36865	CCCC CCCC	25
2. 36866	HDDD DDDD	150
3. 36867	GEEE EEEF	46 oder 176
4. 36868	GGGG GGGG	
5. 36869	HHHH IIII	240
6. 36870	KKKK KKKK	0
7. 36871	LLLL LLLL	0
8. 36872	MMMM MMMM	255
9. 36873	NNNN NNNN	255
10. 36874	JRRR RRRR	0
11. 36875	OSSS SSSS	0
12. 36876	PTTT TTTT	0
13. 36877	QUUU UUUU	0
14. 36878	WWWW VVVV	0
15. 36879	XXXX YZZZ	27

A:	Einblendungsmodus
B:	Horizontale Bildschirmzentrierung
C:	Vertikale Bildschirmzentrierung
D:	Zahl der Bildschirmspalten
E:	Zahl der Bildschirmzeilen
F:	Zeichengröße (8x8 oder 8x16)
G:	Lichtgriffelraster
H:	Bildschirmspeicheradresse
I:	Zeichenspeicheradresse
K:	Lichtgriffel (horizontal)
L:	Lichtgriffel (vertikal)
M:	Paddel (X)
N:	Paddel (Y)
J,O,P,Q:	Tongenerator 1-4 ein/aus
R,S,T,U:	Frequenz Tongenerator 1-4
V:	Lautstärke
W:	Hilfsfarbe
X:	Bildschirmfarbe
Y:	RVS-Modus
Z:	Rahmenfarbe

Tabelle 1. Die einzelnen Registerbelegungen im VIC

Nun wollen wir sehen, wie man die Startadressen der einzelnen VIC-externen Speicherstellen ermitteln kann. Beginnen wir mit dem Videospeicher.

Wenn man Bild 1 betrachtet merkt man schon, wie haarig die Berechnung mit Hilfe der einzelnen Bits ist. Aus den beiden hier beteiligten Registern Nummer (#) 2, Adresse 36865, und Nummer 5 (Adresse 36869) müssen jeweils bestimmte Bits für die Erstellung der Adresse herangezogen werden. Bild 1 zeigt nun, wie die jeweiligen Bits (die vier höchstwertigen aus Register #5 und Bit 7 aus Register #2) in das High-Byte-Schema »eingebaut« werden. Ferner muß Bit 7 aus Register #2 invertiert werden, das heißt aus der Null wird eine Eins und umgekehrt.

Fassen wir noch einmal zusammen: Aus bestimmten Bits der Register 2 und 5 entsteht das High-Byte der Bildschirmspeicheranfangsadresse (ein Low-Byte gibt's hier nicht). Außerdem sind bestimmte Bits aus diesem Adreßschema immer auf Null. Die Konsequenz daraus ist, daß der Videobereich nur in bestimmten Speicherbereichen angesiedelt werden kann. Tabelle 2 zeigt die möglichen Adressen, die sich aus Bild 1 ergeben. Zu beachten ist, daß das untere Nibble (1 Nibble = 4 Bits) im Register #5 dem Zeichenspeicher zugeordnet ist, dieses darf vorläufig noch nicht angetastet werden.

Kompliziertes — Die Adreßermittlung

In gleicher Weise wie für den Bildschirm wird auch die Adresse des Farbspeichers ermittelt.

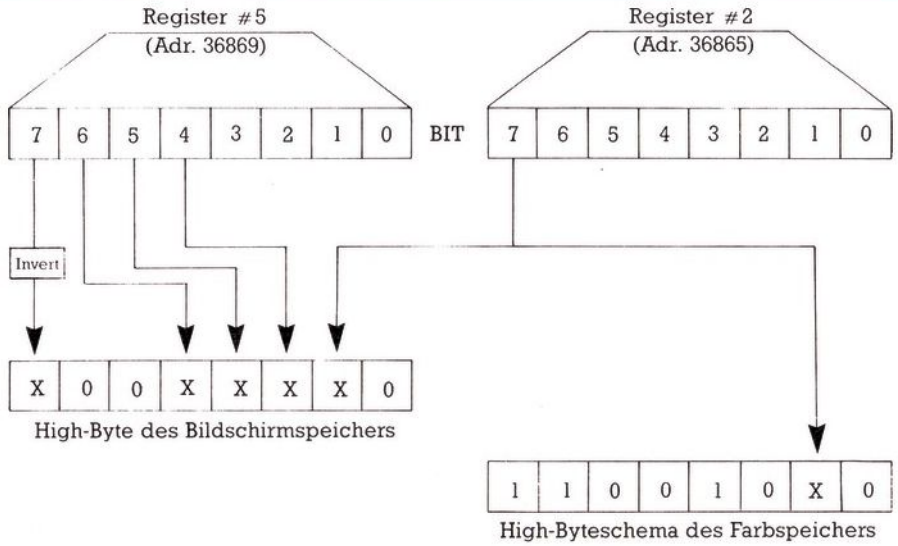

Bild 1. So werden die Bits aus den Registern #2 und #5 in die Adreßschemata eingebaut

Register #5 Bit 4-7	Register #2 Bit 7	Adresse	Bemerkung
1000	0	0	Zeropage (nicht nutzbar)
1000	1	512	Stack (nicht nutzbar)
1001	0	1024	
1001	1	1536	
1010	0	2048	
1010	1	2560	3 KByte Erweiterung
1011	0	3073	
1011	1	3584	
1100	0	4096	
1100	1	4608	
1101	0	5120	
1101	1	5632	Grundversions-
1110	0	6144	speicher
1110	1	6656	
1111	0	7168	
1111	1	7680	

Tabelle 2. Diese Tabelle zeigt die möglichen Bildschirmspeicherstellen

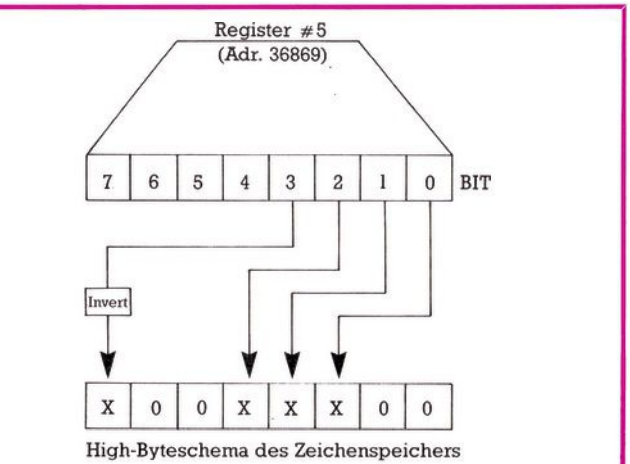

Bild 2. Auch die Anfangsadresse des Zeichengenerators wird mit Hilfe eines Adreßschemas ermittelt

Auch hier gibt es ein Adreßschema (Bild 1), in das Bit 7 aus Register #2 eingesetzt wird. Daher erklärt sich auch, warum sich bei einer Erweiterung von mehr als 8 KByte der Farbspeicher verschiebt. Liegt der Bildschirmspeicher bei Adresse 4096, dann hat Bit 7 aus Register #2 den Wert Null, wodurch sich eine Coloradresse von 37888 ergibt. Im anderen Fall (Bildschirmstartadresse 7680) ist genau dieses Bit auf eins, wodurch es eine Verschiebung nach 38400 gibt.

Kommen wir nun zum Zeichenspeicher. Er erhält Informationen über jedes Zeichen, das auf dem Bildschirm darstellbar ist. Auch hier kann über das untere Nibble von Register #5 die Startadresse ermitteln. (Bild 2). Wenn die Bits 0-3 auf Null gesetzt sind, liegt sie bei 32768, also im Zeichengenerator-ROM (Normalstellung). Dort sind alle beim VC 20 verfügbaren Zeichen abgelegt (in welcher Form dies geschieht, werden wir später noch sehen).

Wie sie bestimmt schon bemerkt haben, spielt bei der Verwaltung des Bildschirms die Adresse 36869 (Register #5) eine bedeutende Rolle. Fragt man sie mit PEEK ab, so erhalten wir in der Grundversion oder bei einer 3-KByte-Erweiterung den Wert 240, bei einem um 8 KByte erweiterten Speicher 192. Zerlegt man diese Zahlen wie in Bild 2, so erhält man jedes Mal die gleiche Adresse für den Zeichenspeicher. Setzen wir nun Bit 1 auf 1 (= 242 in GV beziehungsweise 194 bei 8 KByte), liegt die Anfangsadresse des Zeichenspeichers bei Adresse 34816 (Tabelle 3), wodurch wir auf einmal Groß- und Kleinschrift auf dem Bildschirm haben. Die eine der zwei zu Anfang gestellten Fragen haben wir damit beantwortet.

Wie sind die im Zeichengenerator abrufbaren Zeichen eigentlich aufgebaut? Spätestens hier müssen wir ein Zeichen (im wahrsten Sinne des Wortes) unter die Lupe nehmen (Bild 3). Die vergrößerte Darstellung läßt erkennen, daß jedes Zeichen aus acht Zeilen mit je acht Spalten (8 x 8 Matrix) zusammengesetzt ist. Innerhalb dieses Gitters sind einzelne Punkte gesetzt beziehungsweise gelöscht. Jede Zeile innerhalb dieser Matrix ist als Byte (mit verschieden gesetzten Bits) im Zeichengenerator-ROM abgelegt (ROM natürlich deshalb, damit die Zeichen nach dem Abschalten erhalten bleiben).

Punkt für Punkt — der Zeichenaufbau

Im unteren Teil von Bild 3 sieht man sehr gut, daß ein gesetzter Punkt der 1, ein nicht gesetzter (gelöschter) Punkt der 0 entspricht. Die Zeile wird dann in einen normalen Zahlenwert zwischen 0 und 255 umgerechnet. In dieser Form und in der Zeilenreihenfolge von oben nach unten sind sie nun im ROM abgelegt. Der »Klammeraffe« @ (Bildschirmcode 0) ist als erstes Zeichen dort zu finden. Folglich enthält die Startadresse des Zeichengenerators (Adresse 32768) die 0-te Zeile des Zeichens — also 28, Adresse 32769 enthält 34 und so weiter. Damit machen wir folgende Rechnung auf:

Zeile	7.	6.	5.	4.	3.	2.	1.	0.	BIT
0.	0	0	0	1	1	1	0	0	= 28
1.	0	0	1	0	0	0	1	0	= 34
2.	0	1	0	0	1	0	1	0	= 70
3.	0	1	0	1	0	1	1	0	= 86
4.	0	1	0	0	1	1	0	0	= 76
5.	0	0	1	0	0	0	0	0	= 32
6.	0	0	0	1	1	1	1	0	= 30
7.	0	0	0	0	0	0	0	0	= 0

Bild 3. Diese Grafik zeigt den Aufbau eines VC 20-Zeichens innerhalb der 8 x 8 Matrix

Ein Zeichen: 8 Byte
Anzahl: 128 Zeichen
Invers: 128 Zeichen

Also 256 Charakter pro Zeichensatz mal 8 Bytes = 2048 Bytes Platzbedarf für Großbuchstaben und Grafikzeichen mit deren inversen Gegenstücken. Dazu kommen nocheinmal so viele Bytes für den zweiten Zeichensatz, womit sich ein Speicherbedarf von 4 KByte für den gesamten Zeichenvorrat des VC ergibt.

Die Platznummern

Jedes Zeichen hat nun eine (eben durch die Reihenfolge im ROM) bestimmte Platznummer zugewiesen bekommen. Der Klammeraffe (@) hat die 0, der Buchstabe A die 1, B die 2 und so weiter.

Diese Platznummern stehen als Bildschirmcodes im Handbuch auf Seite 141. POKEt man nämlich ein Zeichen ins Bildschirm-RAM, so multipliziert der VIC den Bildschirmcode mit 8 und addiert die Startadresse des Zeichengenerators hinzu.

Beispiel: Das Zeichen $ (Bildschirmcode 36) ist ab folgender Adresse abgespeichert (Formel 1).

AD = BC * 8 + ZG
36 * 8 + 32768
33056

Diese Formel müssen wir im Hinterkopf behalten, denn wir benötigen sie später noch.

Weiterhin ist anzumerken, daß jeweils nur 256 Zeichen gleichzeitig dargestellt werden können (eine Ausnahme davon lernen wir nächstes Mal kennen).

Register #5 Bits 0-3	Adresse	POKE 36869, (GV)	(8KByte)	Belegung
0000	32768	240	192	Zeichen ROM
0001	33792	241	193	Inverse Großschrift
0010	34816	242	194	Groß-, Kleinschrift
0011	35840	243	195	Inverse Kleinschrift
0100	36864	244	196	
0101	37888	245	197	Nicht verwendbar, da belegt
0110	38912	246	198	
0111	39936	247	199	
1000	0	248	200	Zeropage
1001	1024	249	201	
1010	2048	250	202	3 KByte Erweiterung
1011	3072	251	203	
1100	4096	252	204	
1101	5120	253	205	Grundversion
1110	6144	254	206	
1111	7168	255	207	

Tabelle 3. Die möglichen Zeichengeneratoranfangsadressen (jedoch sind nicht alle möglichen Adressen auch sinnvoll)

Characterstyling: Eigene Sonderzeichen

Schön wäre es nun, wenn man jedes Zeichen nach eigenen Wünschen abändern könnte. Dies geht (wie sich inzwischen wohl herumgesprochen hat), indem man den Zeichengenerator (also die Zeicheninformationen) ins RAM verlegt und das Register #5 ändert.

Dazu müssen wir uns anhand einer Grafik des Speicheraufbaus (Bild 4) überlegen, wo wir diese Zeichen abspeichern, denn wie in Tabelle 2 zu sehen ist, lassen sie sich nicht überall, sondern nur an bestimmten Stellen unterbringen. Auch hier muß man zwischen den beiden Aus-

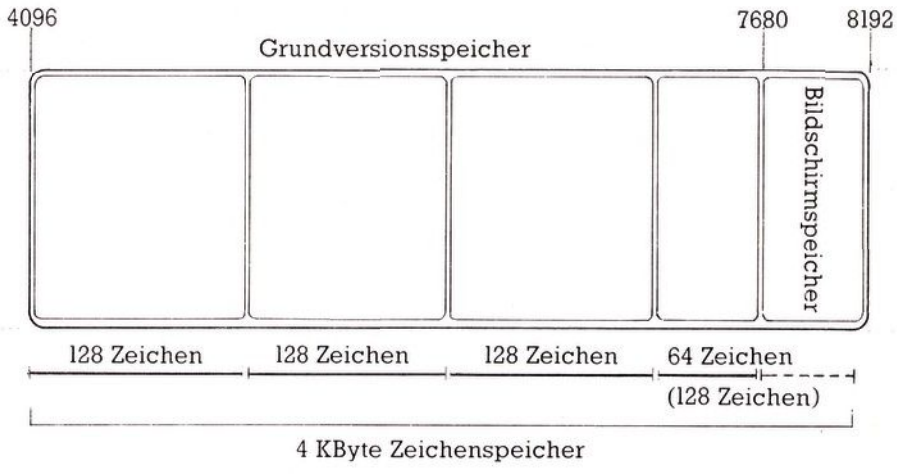

Bild 4. So können die 512 Zeichen im Grundversions-RAM verteilt werden

baustufen Grundversion /+3 KByte und ab 8 KByte unterscheiden.

Nachhilfe — Die logischen Funktionen

Aufgrund der Zweiteilung des Registers #5 sollte man alle Änderungen nur mit Hilfe logischer Funktionen durchführen, dazu ein kleiner Exkurs.

Gerade bei unserem Fall, aber auch bei anderen Gelegenheiten, will man nur bestimmte Bits einer Speicherstelle ändern. Mit den boolschen Operationen OR und AND lassen sich jeweils die gewünschten Bits Ein- beziehungsweise Ausschalten.

Die erste wichtige Verknüpfungsoperation ist die ODER- (in Basic OR) Operation, die zwei Bits nach der in Tabelle 4a abgebildeten Wahrheitstabelle verknüpft. Haben wir nun eine Kette von 8 Bit (also ein Byte), so kann man mit Hilfe dieser Operation bestimmte Bits »einschalten«. Dazu ein Beispiel: Von einer Speicherstelle sollen die Bits 1 und 6 eingeschaltet werden. Da Basic die Binärzahlen nicht direkt, sondern nur als Dezimalzahlen verarbeiten kann, müssen wir zunächst alle Werte in dieses Zahlensystem umwandeln.

Bit	7	6	5	4	3	2	1	0
Wertigkeit	128	64	32	16	8	4	2	1
	0	1	0	0	0	0	1	0

Durch Addition der Wertigkeiten der zu setzenden Bits ergibt sich der Wert 66 (= 64+2). Durch die OR-Verknüpfung der Speicherstelle mit 66 ergibt sich folgende Bitstruktur (für den ursprünglichen Inhalt der Speicherstelle nehmen wir einmal 229 an):

Speicherstelle	1	1	1	0	0	1	0	1	=229
Bit 1+6 setzen:	0	1	0	0	0	0	1	0	=66
229 OR 66:	1	1	1	0	0	1	1	1	=231

Der gewünschte Effekt ist eingetreten, das heißt die Bits 1 und 6 wurden gesetzt (da Bit 6 bereits eingeschaltet war, ergab sich hier keine Änderung).

Analog verfährt man beim Löschen bestimmter Bits. Dazu wird dann allerdings die UND-Operation (Basic-Befehl AND) verwendet. Wie Tabelle 4b zeigt, bleiben nur die Bits unverändert, die mit 1 »verANDet« werden. Auch hierzu ein Rechenbeispiel, bei dem die Bits 1 und 6 gelöscht werden sollen:

Speicherstelle:	1	1	1	0	0	1	1	1	=231
Bit 1+6 löschen:	1	0	1	1	1	1	0	1	=189
231 AND 189:	1	0	1	0	0	1	0	1	=165

Auch hierbei muß man sich vor der Verknüpfung ein Bitmuster berechnen, in dem die zu löschenden Bits auf Null, die nicht zu verändernden auf 1 gesetzt werden müssen. Wem dies noch nicht restlos klar ist, der sollte anhand von einigen Rechenbeispielen diese Operationen üben, denn wir benötigen diese in der nächsten Folge, wenn es darum geht, jeden Grafikpunkt einzeln anzusteuern (beispielsweise über Koordinaten).

Aber auch das Register #5 kann damit wesentlich eleganter geändert werden. So schaltet POKE 36869,PEEK (36869) OR 1 auf Groß-/Kleinschreibung um (überlegen Sie mal, warum dies so ist), wobei man keine Unterscheidung zwischen verschiedenen Speicherausbaustufen treffen muß.

Fassen wir das bis jetzt behandelte noch einmal kurz zusammen: Wir haben also die Möglichkeit, den gesamten Zeichensatz ins RAM zu verlegen. Dort kann er dann nach eigenen Wünschen verändert werden. Bevor man jedoch irgendeine Änderung vornehmen kann, müssen die Zeichen in das bisher ja »leere« RAM kopiert werden.

Das in Listing 1 abgedruckte Basic-Programm erfüllt diese Aufgabe. Die Routine ist so aufgebaut, daß lediglich die Variable RG eingesetzt werden muß. Sie wurde mit Absicht so allgemein gehalten, damit sie für alle Speichererweiterungen einsetzbar ist.

Hier nun eine Auflistung der einsetzbaren Werte (sie entsprechen im übrigen denen in Tabelle 2):

RG = 12: Aus diesem Registerwert errechnete sich das Programm die Zeichenspeicheradresse 4096. Die Zeichen füllen bei dieser Einstellung also den gesamten Grundversionsspeicher aus (aber wer benötigt schon 512 verschiedene Zeichen?). Man nutzt diese Zeichenfülle also nur bei einem um mindestens 3 KByte erweiterten Speicher aus. In diesem Fall (und auch bei einer 8-KByte-Erweiterung) darf man den Bildschirmspeicher nicht außer acht lassen, denn dieser befindet sich ja auch im Bereich zwischen 4096 und 8192 (je nach Erweiterung). In beiden Fällen ergeben sich also Überschneidungen zwischen den beiden Speichern. So können bei einem 8-KByte-Speicher die unteren 512 Byte nicht für Zeicheninformationen verwendet werden (da dort ja der Bildschirmspeicher liegt). Also reduziert sich die Anzahl der verfügbaren Zeichen um 64.

Bei einer 3-KByte-Erweiterung wird man mit dem gleichen Problem konfrontiert. In diesem Fall werden die obersten 512 Bytes vom Videospeicher in Beschlag genommen.

ZG = 13 (Startadresse 5120): Mit diesen 384 Zeichen kommt man in der Regel gut aus (wie man alle auf den Bildschirm bringt, sehen wir in der nächsten Folge). Bezüglich der Nutzbarkeit der Zeichen ergibt sich nur bei der Grundversion beziehungsweise 3-KByte-Erweiterung das Problem, daß der obere Bereich (also Adresse 7680 bis 8192) vom Bildschirmspeicher belegt werden. Das betrifft die Bildschirmcodes 64 bis 128 im Klein-/Großschriftmodus.

ZG = 14 (Startadresse 6144): Hier haben wir Platz für 256 Zeichen, die bei 8 KByte voll nutzbar sind. Im anderen Fall sind wiederum 64 Zeichen für den Videospeicher zu subtrahieren.

ZG = 15 (Startadresse 7168): Diese Zeichenmenge wird üblicherweise in der Grundversion verwendet. Hierbei reduziert sich der Programmspeicher nämlich nur um 512 Byte (also 64 Zeichen). Die anderen 64 Zeichen fallen ja wiederum dem Bildschirmspeicher zum Opfer.

Da das Zeichenkopieren fast immer notwendig ist, verwendet man anstelle des Basicprogramms aus Geschwindigkeitsgründen besser eine Maschinenroutine (Listing 2 und 3). Das Programm kann mit Hilfe des Laders in ihre eigene Grafikrouti-

```
100 rem ********************************
110 rem *** dieses programm kopiert ***
120 rem *** zeichen aus dem rom ins ***
130 rem *** ram. die  routine  kann ***
140 rem *** in ihr eigenes grafik - ***
150 rem *** programm eingebaut wer- ***
160 rem *** den. lediglich die va - ***
170 rem *** riable  rg  muss  der   ***
180 rem *** routine  uebergeben     ***
190 rem *** werden.                 ***
200 rem ********************************
210 fort=0to48:readd:poket+828,d:s=s+d
220 next:rem *** daten einlesen
230 if s<>4890 thenprint"datenfehler !":
end
240 rg =14 : rem *** uebergabevariable
250 poke4,rg
260 sys828
270 data169,000,133,000,133
280 data002,169,128,133,001
290 data165,004,056,233,008
300 data010,010,133,003,166
310 data004,160,000,177,000
320 data145,002,230,000,230
330 data002,208,246,230,001
340 data230,003,202,208,239
350 data173,005,144,005,004
360 data141,005,144,096

ready.
```

Listing 2. Zeichensatz kopieren (Basic-Lader)

```
****************** character copy
033c  lda #$00    ; pointer init
033e  sta $00
0340  sta $02
0342  lda #$80    ; zeiger auf zeichen-
0344  sta $01     ; generator rom
0346  lda $04     ; rg variable holen
0348  sec
0349  sbc #$08    ; umrechnung in h-byte
034b  asl         ; multiplikation mit 4
034c  asl
034d  sta $03
034f  ldx $04     ; schleifenzaehler
0351  ldy #$00
0353  lda ($00),y; zeichen laden
0355  sta ($02),y; im ram abspeichern
0357  inc $00     ; low-byte inc.
0359  inc $02
035b  bne $0353   ; uebertrag ?
035d  inc $01     ; ja, dann high-byte
035f  inc $03     ; inkrementieren
0361  dex         ; zaehlschleife
0362  bne $0353
0364  lda $9005   ; auf ram-zeichen um-
0367  ora $04     ; schalten
0369  sta $9005
036c  rts
```

Listing 3. Zeichensatz kopieren (Assembler-Darstellung)

```
10 rg=14:zg=(rg-8)*1024
20 fort=0to8191-zg
30 pokezg+t,peek(32768+t)
40 next
50 poke36869,peek(36869)orrg

ready.
```

Listing 1. Zeichensatz ins RAM kopieren

Wert 1	Wert 2	Verknüpfung
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 4a. Die Wahrheitstabelle der ODER-Operation

Wert 1	Wert 2	Verknüpfung
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 4b. Die Wahrheitstabelle der UND-Operation

ne eingebaut werden. Auch speicherplatzmäßig ergeben sich, da das Maschinenprogramm recht kurz ist, keine Probleme, denn es liegt im Bandpuffer (Adresse 828).

Das Ändern der Sonderzeichen

Nun sind wir an einer Stelle angelangt, von der aus wir die Zeichen nach eigenen Wünschen abändern können, seien es mathematische Sonderzeichen, deutsche Umlaute oder Grafikzeichen für eigene Spielprogramme.

Am Anfang steht der Entwurf eines Zeichens mit Hilfe einer 8 x 8 Matrix (Bild 5). Hier in unserem Beispiel soll das Pfundzeichen £ (Bildschirmcode 28) durch das Copyrightzeichen ersetzt werden.

Dazu wird unser Beispielprogramm aus Listing 1 um folgende 5 Zeilen ergänzt:

```
60 CH = 28: AD = CH *8 + ZG:
REM Formel #1
 70 FOR Z = 0 TO 7
 80 READ D
 90 POKE AD*Z,D: NEXT
100 DATA 60, 66, 153, 161, 153, 66,
60
```

Der Bildschirmcode des Zeichens wird durch die Variable CH übergeben. Daraus errechnet sich das Programm die Startadresse des Zeichens, indem es den Bildschirmcode mit 8 multipliziert und die Anfangsadresse des Zeichenspeichers dazuaddiert (in unserem Fall 6144).

Das Abspeichern der Zeichen

Im allgemeinen erstellt man solche Zeichen nicht aus Spaß an der Freude, sondern man möchte sie in Spielen oder anderen Programmen verwenden. Dazu müssen sie in irgendeiner Form abgespeichert werden. Hierfür gibt es wieder zweierlei Möglichkeiten. Entweder man schreibt die Zeicheninformationen in Form von DATA-Zeilen ins Programm oder man SAVEt sie direkt auf Band oder Diskette ab.

	7.	6.	5.	4.	3.	2.	1.	0.	Bit	
0.	0	0	1	1	1	1	0	0	=	60
1.	0	1	0	0	0	0	1	0	=	66
2.	1	0	0	1	1	0	0	1	=	153
3.	1	0	1	0	0	0	0	1	=	161
4.	1	0	1	0	0	0	0	1	=	161
5.	1	0	0	1	1	0	0	1	=	153
6.	0	1	0	0	0	0	1	0	=	66
7.	0	0	1	1	1	1	0	0	=	60
Zeile										

Bild 5. Auch die eigenen Zeichen muß man mit Hilfe der Matrix entwerfen (hier als Beispiel das Copyright-Zeichen)

Beide Verfahrensweisen eignen sich für bestimmte Anwendungsgebiete besonders gut, für andere weniger gut.

Die DATA-Zeilen-Methode eignet sich dann, wenn es darum geht, lediglich 3 oder 4 Zeichen abzuändern (beispielsweise für ein Textverarbeitungsprogramm mit deutschen Umlauten). Diese werden dann — wie in dem Beispiel oben — als DATA-Zeilen ins Programm geschrieben.

Für größere Änderungen am Zeichenvorrat ist diese Methode aber aus Geschwindigkeits- und Platzgründen (sie benötigt etwa viermal so viel Platz) denkbar ungeeignet.

Bei der anderen Methode werden die Zeicheninformationen wie ein Programm einfach auf Band abgespeichert. Dazu besinnen wir uns wieder auf Folge 1 in Ausgabe 9, wo von den Basic-Zeigern die Rede war. Ferner ist dort beschrieben, wie man Platz für Maschinenprogramme und Sonderzeichen schafft, und sie vor dem Zugriff des Basicinterpreters schützt. Auch hier muß man — wie so oft, wenn es um Grafik oder Bildschirm geht — zwischen den zwei grundsätzlichen Ausbauversionen unterscheiden.

All diese Vorgänge möchte ich anhand von Bild 6 erklären (6a für die Grundversion, 6b für die 8-KByte-Erweiterung):

Grundversion (Bild 6a): Der dunkelgelb unterlegte Teil der Speichergrafik stellt den Adreßbereich dar, auf den der Interpreter zurückgreift. Dieser geht normalerweise bis Adresse 7680 (zu erkennen an der hellgelben Farbe).

Durch die Umstellung mit POKE 55,0: POKE 56, 28:CLR hat man Platz für 64 Zeichen (= 512 Byte) geschaffen, die vom Interpreter nicht angetastet werden (dies haben wir ja in Folge 1 schon besprochen).

Nun aber zu dem Abspeichern des Zeichensatzes. Auch beim SAVEn richtet sich der VC nach diesen Zeropagezeigern, denn er speichert alles ab, was er zwischen den beiden Adreßpaaren 43, 44 (Basic-Anfang) und 45,45 (Programmende) findet. Folglich gibt das erste Paar die Anfangs-, das andere die Endadresse der zu speichernden Daten an. Normalerweise sind

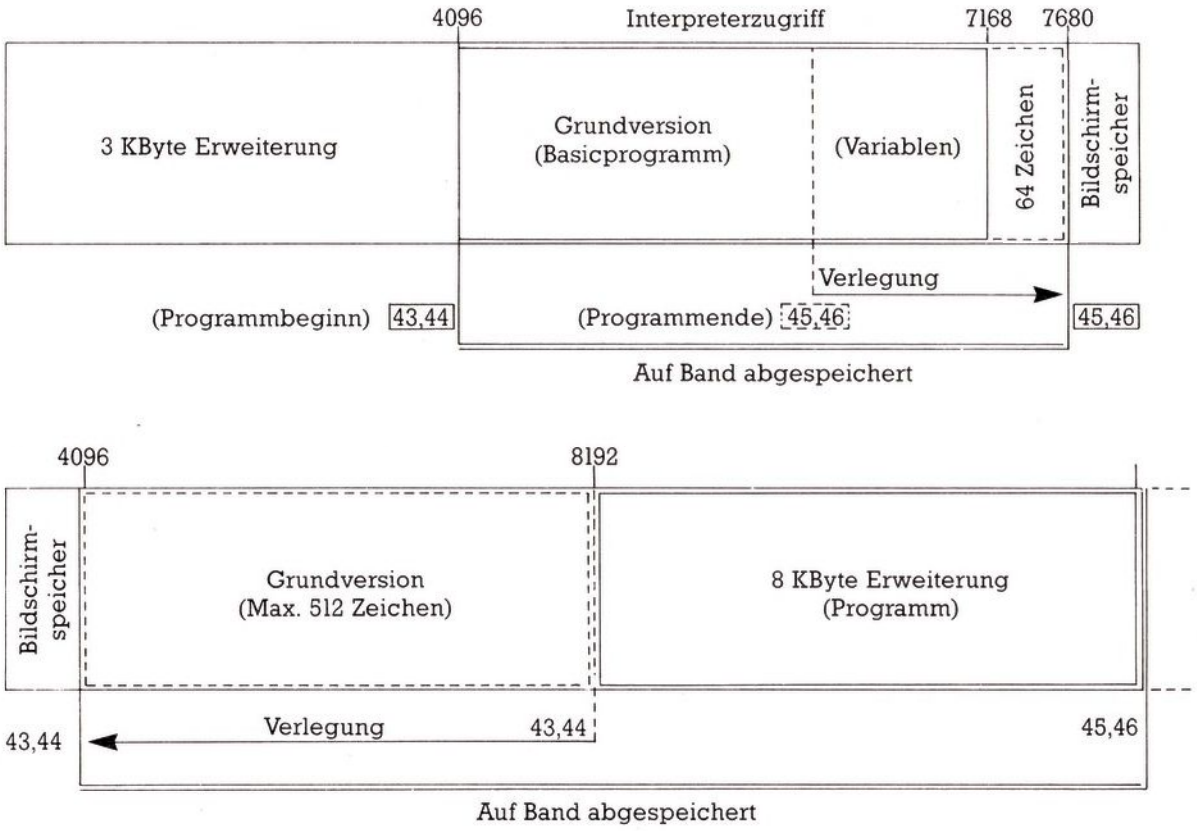

Bild 6. Durch die Verlegung der Basic-Zeiger SAVEt der Computer auch die Sonderzeichen mit ab: (a) bei Grundversion / 3-KByte-Erweiterung, (b) ab 8 KByte Speicherausbau

diese Daten das Basic-Programm, welches sich zwischen diesen Zeigern befindet. Durch eine Änderung der beiden Zeropagespeicherstellen 45 und 46 auf das Ende des Zeichensatzs (Adresse 7680) bewirkt man, daß das gesamte Programm mit Variablenbereich und Zeichensatz abgespeichert wird (Bild 6a — rote Markierung).

Schritt für Schritt auf Band

Hier noch einmal die nötigen Programmschritte:

1. Programm um Zeile 5 ergänzen.

5 POKE 45, XXX: POKE 46, XX: POKE 55, 0: POKE 56, 28: CLR

Warum muß das Programm um eine Zeile ergänzt werden? Nun, da sich der Computer bei der Variablenverwaltung nach diesen Zeigern richtet, müssen diese nach dem Ladevorgang wieder auf den alten Wert — der auf das wirkliche Programmende zeigt — gesetzt werden, damit der Computer nicht mit der Verwaltung durcheinander kommt. Auch die Zeichen selbst müssen nach dem Laden wieder vor dem Interpreterzugriff geschützt werden, welches durch die Veränderung der Zeiger 55, 56 geschieht.

2. Nun stellen wir quasi »zu Fuß« den Inhalt von Adresse 45 und 46 fest:

PRINT PEEK(45), PEEK(46)

3. Diese Werte werden nun nachträglich (statt »xxx«) in Zeile 5 geschrieben, wobei zu beachten ist, daß der Wert von Adresse 45 ein-, zwei- oder dreistellig sein kann (oben wurden drei Stellen angenommen). Sollten es nun nicht drei Ziffern sein, so ist der Rest mit Nullen zu ergänzen (zum Beispiel 2 = 002 oder 34 = 034). Wichtig ist, daß sich die Zeilenlänge nicht ändert, da sich damit auch das Programmende verschieben würde.

4. Die Zeiger werden nun zum Abspeichern vorbereitet: POKE 45, 0: POKE 46, 30: CLR: SAVE»...«.

Wie man sieht, befindet sich das Programm vor den Sonderzeichen im Speicher. Die Zeiger müssen zum Abspeichern also nach oben gesetzt werden. Bei einer 8-KByte-Erweiterung liegt der Fall genau anders herum. Dort liegt der Zeichensatz nämlich vor dem Programm. Folglich muß hier der Basic-Anfang beim Abspeichern nach unten gelegt werden, aber dies besprechen wir jetzt im Folgenden noch genauer:

8-KByte-Erweiterung (Bild 6b): Auch hier ist die erste Handlung das Verstellen eines Basic-Zeigers. Auffällig ist, daß die Sonderzeichen — wie oben bereits angesprochen — nicht mehr oberhalb des Basic-Programms (also wie in Bild 6a zu sehen an dessen Ende), sondern unterhalb liegen. Daher muß die Basic-Anfangsadresse so geändert werden, daß auch hier kein zerstörendes Eingreifen mehr möglich ist. Durch das Hochsetzen der Startadresse von 4608 auf 8129 wird dies erreicht: POKE 44, 32: POKE 8192,0: NEW

Diese Anweisung ist vor dem Laden oder der Eingabe des Programms notwendig, denn wie man sieht, muß nach der erstmaligen Umschaltung der Programmspeicher mit NEW gelöscht werden.

Nachdem sich nun das Programm und die Sonderzeichen im Speicher befinden, können beide zusammen wieder abgeSAVEt werden; auch dazu ein »Rezept«:

1. Den Inhalt von Zeigerpaar 45,46 feststellen und notieren.
2. Basic-Anfang auf 4608 (Ursprungswert) zurückstellen:
POKE 44,18: NEW
3. Zeile eingeben:
10 POKE 44,32: RUN
4. Den notierten Zeigerinhalt in die beiden Zeropagespeicherstellen zurückschreiben:
POKE 45,Low-Byte: POKE 46, High-Byte: CLR
5. Die Zeichen mit dem Programm abspeichern (SAVE).

Lädt man das Programm wieder in den Speicher, so muß das High-Byte des Basic-Anfangszeigers (44) auf 32 gestellt werden, denn dort befindet sich ja das eigentliche Programm. Zeile 10 in Schritt 3 hat diese Aufgabe. Die Zeile befindet sich am zurückgestellten Programmbeginn (4608), also noch vor dem Zeichensatz, der erst bei Adresse 5120 beginnt. Das hat den Vorteil, daß man das Programm direkt mit RUN starten kann.

Soweit der wichtige Abschnitt über das Abspeichern der Sonderzeichen, der mit Absicht etwas umfangreich ausgefallen ist, denn auch derjenige, der nicht so viele Kenntnisse über den VC 20 hat, soll in der Lage sein, seine grafischen Werke auf Band zu bringen.

Mit dieser Erkenntnis beschließen wir die Einführung in die Grafikfähigkeit des VC 20. Das nächste Mal benutzen wir die bisher gewonnenen Grundlagen, um voll in die Materie einzusteigen.

(Christoph Sauer/ev)