

Sprites

Nach soviel grauer Theorie kommt nun wieder Bewegung ins Spiel!

In diesem Kapitel geht es um die Programmierung von Sprites. Das sind kleine, bewegliche Objekte, deren Aussehen Sie innerhalb bestimmter Grenzen frei gestalten können und die sich dann beispielsweise für Spiele verwenden lassen.

Wir werden die Sprite-Programmierung zunächst in BASIC durchführen, um die grundlegenden Abläufe kennenzulernen. Aber keine Sorge, für jedes BASIC-Programm werden wir immer das entsprechende Assembler-Gegenstück erstellen.

Sprites werden vom Commodore 64 bereits seitens der Hardware unterstützt und das vereinfacht die Programmierung erheblich. Es werden standardmäßig 8 Sprites unterstützt, doch es sind durch Anwendung spezieller Techniken auch mehr Sprites möglich.

Es gibt einfarbige Sprites und mehrfarbige Sprites, wobei wir uns zunächst mit den einfarbigen Sprites beschäftigen wollen.

Einfarbige Sprites können eine von 16 Farben annehmen und maximal 24 Pixel breit bzw. maximal 21 Pixel hoch sein. Ein Sprite besteht also insgesamt aus 504 Punkten.

Bevor wir mit einem Sprite arbeiten können, müssen wir erst einmal wissen, wie es aussehen soll.

Doch wie sagt man dem C64, wie man sich das Sprite vorstellt?

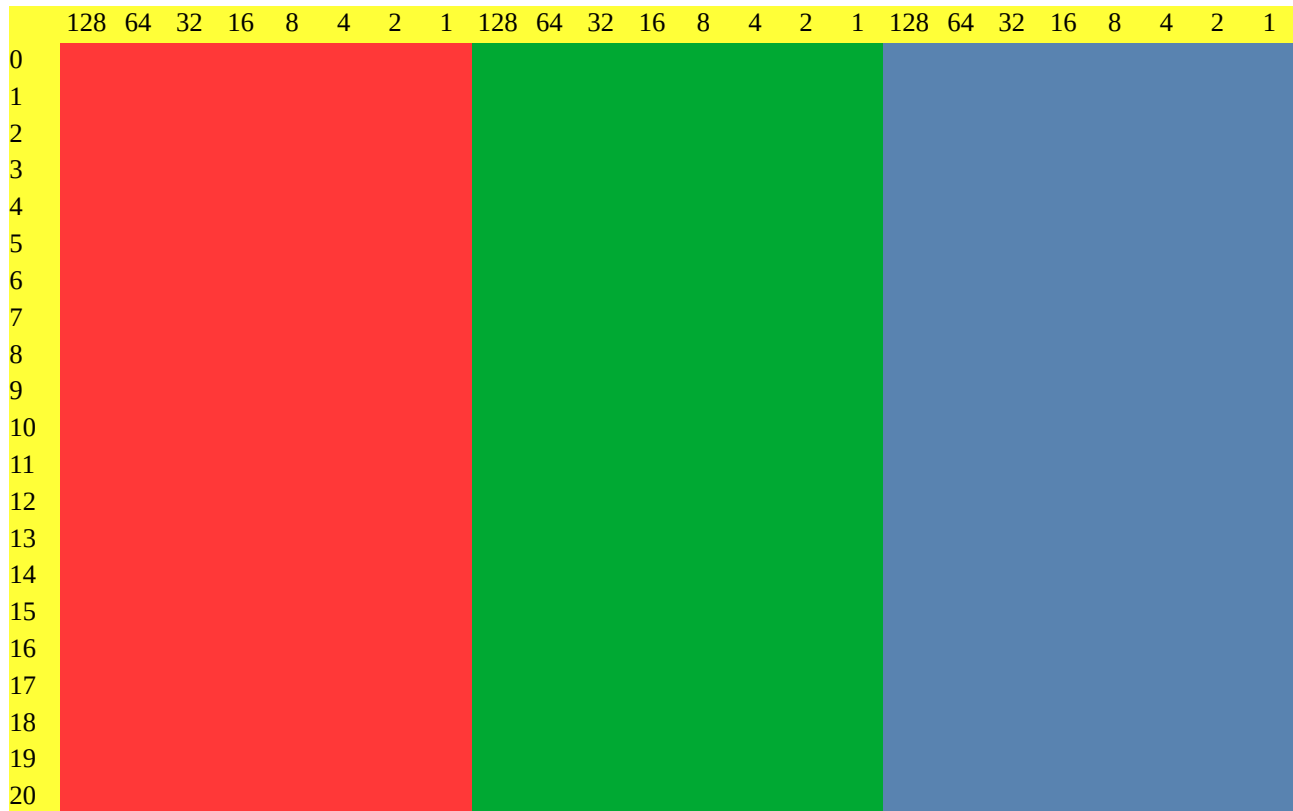
Dazu zeichnet man sich zunächst beispielsweise auf kariertem Papier einen Raster mit 24 Spalten und 21 Zeilen auf, wobei jede Zelle des Rasters einem der 504 Pixel des Sprites entspricht.

Das Sprite hat eine horizontale Auflösung von 24 Pixel und wenn man jedem Pixel ein Bit zuordnet, dann benötigen wir 3 Bytes ($3 \times 8 \text{ Bit}$ für 24 Pixel) um eine Zeile aus unserem Raster speichern zu können.

In vertikaler Richtung beträgt die Auflösung 21 Pixel, d.h. wir benötigen insgesamt ($3 \times 21 \text{ Bytes} = 63 \text{ Bytes}$) um das Aussehen unseres Sprites festzulegen.

Ein Block mit Spritedaten muss jedoch 64 Bytes umfassen, daher folgt auf das letzte Byte noch ein Platzhalter-Byte zum nächsten Block.

Unser Sprite-Raster sieht folgendermaßen aus:



Jede Zeile besteht wie gesagt aufgrund der horizontalen 24 Pixel aus 3 Bytes, der rote Bereich entspricht dem ersten, der grüne Bereich dem zweiten und der blaue Bereich dem dritten Byte in jeder Zeile.

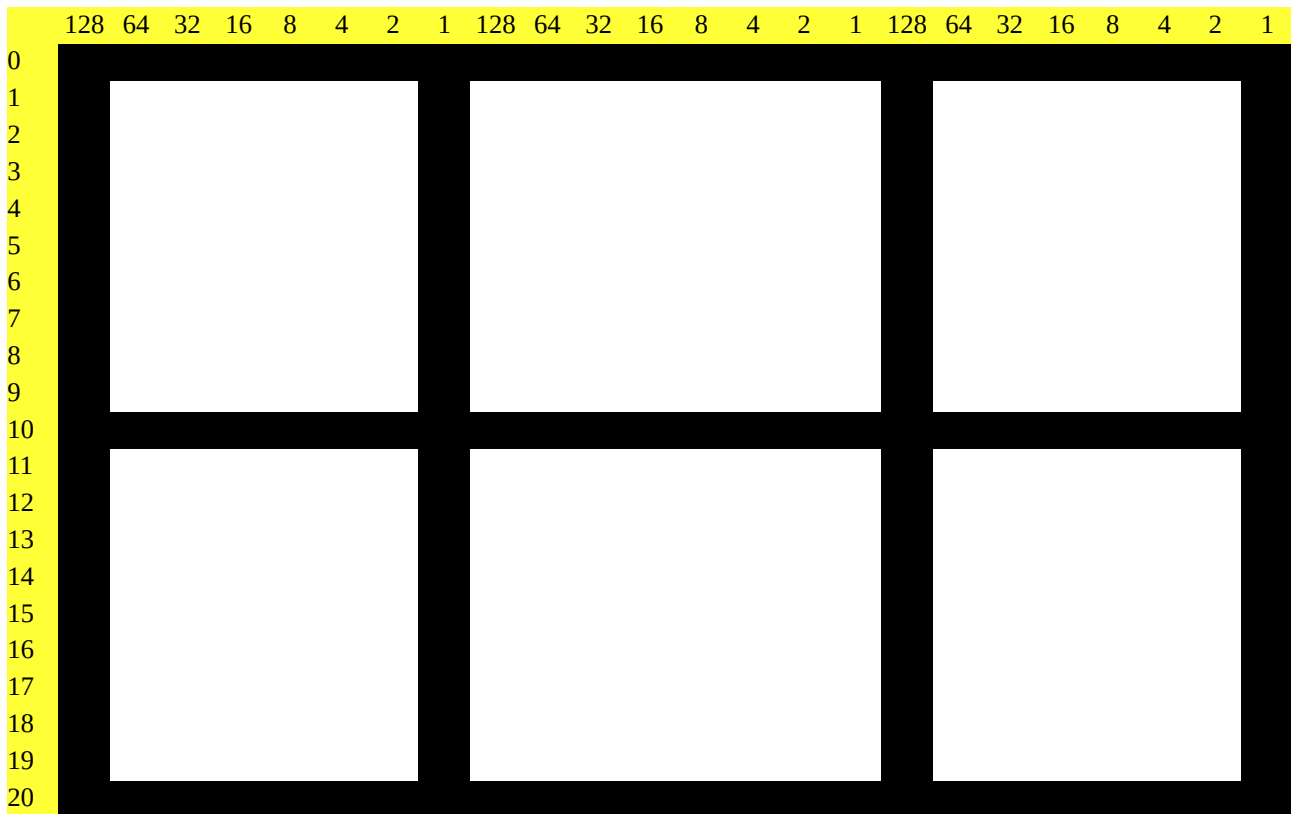
Über jedes Bit der drei Bytes schreiben wir die jeweilige Wertigkeit an der Stelle.

Diese beginnt jeweils mit 128 (2^7) und endet jeweils mit 1 (2^0)

Nehmen wir nun einen leeren Raster und zeichnen uns Pixel für Pixel ein einfach gehaltenes Sprite.

Wir füllen alle Stellen im Raster, an die wir einen Pixel setzen wollen.

Zeichnen Sie folgende einfache Form in den Raster. Jede ausgefüllte Rasterzelle entspricht einem gesetzten Pixel (das Bit hat also den Wert 1). An den weißen Stellen haben wir keinen Pixel gesetzt (das Bit hat also den Wert 0), d.h. hier scheint der Hintergrund durch.



Sehen wir uns das erste Byte in der ersten Zeile an, hier haben wir an jeder Bitposition eine ausgefüllte Zelle, also eine 1. Dies entspricht der binären Zahl %11111111 (hexadezimal \$FF bzw. dezimal 255)

Beim zweiten und dritten Byte ist ebenfalls an jeder Bitposition eine 1, d.h. wir haben auch hier den binären Wert %11111111 (hexadezimal \$FF bzw. dezimal 255)

Unsere erste Zeile wird also durch die drei Bytes 255,255,255 beschrieben.

Gehen wir nun zum ersten Byte in der zweiten Zeile.

Hier haben wir an den Bitpositionen 7 und 0 eine 1 stehen, d.h. wir haben hier die binäre Zahl %10000001 (hexadezimal \$81 bzw. dezimal 129)

Im zweiten Byte haben wir keine gesetzten Bits, d.h. wir haben hier den binären Wert %00000000 (hexadezimal \$00 bzw. dezimal 0)

Das dritte Byte entspricht dem ersten Byte, auch hier haben wir den binären Wert %10000001 (hexadezimal \$81 bzw. dezimal 129)

Die zweite Zeile wird also durch die drei Bytes 129,0,129 beschrieben.

Das setzen wir nun fort bis zur letzten Zeile und erhalten insgesamt folgende Zahlenwerte für die 21 Zeilen:

Erstes Byte	Zweites Byte	Drittes Byte
255	255	255
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
255	255	255
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
129	0	129
255	255	255

Soweit so gut. Aber wo speichern wir diese Zahlen nun ab? Da wir wie gesagt zunächst in BASIC programmieren wollen, legen wir die Zahlen in DATA-Zeilen ab.

Wir beginnen mit hohen Zeilennummern, da wir davor später noch weiteren BASIC-Code einfügen wollen.

Hinweis:

Das Programm befindet sich unter dem Namen SPRITE1BAS auf der Diskette falls Sie es nicht selber abtippen möchten.

```

READY.
1000 REM DATEN FUER SPRITE 0
1010 DATA 255,255,255
1020 DATA 129,0,129
1030 DATA 129,0,129
1040 DATA 129,0,129
1050 DATA 129,0,129
1060 DATA 129,0,129
1070 DATA 129,0,129
1080 DATA 129,0,129
1090 DATA 129,0,129
1100 DATA 129,0,129
1110 DATA 255,255,255
1120 DATA 129,0,129
1130 DATA 129,0,129
1140 DATA 129,0,129
1150 DATA 129,0,129
1160 DATA 129,0,129
1170 DATA 129,0,129
1180 DATA 129,0,129
1190 DATA 129,0,129
1200 DATA 129,0,129
1210 DATA 255,255,255

```

Nun müssen wir diese Daten an einem passenden Platz im Speicher ablegen.
Aber wo? Und wie sagen wir dann dem C64 wo wir die Daten für unser Sprite abgelegt haben?

Kümmern wir uns zuerst darum, wo wir unsere Daten im Speicher ablegen.

Sprite-Daten können wir nicht an jeder beliebigen Stelle im Speicher ablegen. Der Speicherbereich, den wir uns aussuchen, muss zwei Kriterien erfüllen:

- Er muss an einer durch 64 teilbaren Adresse beginnen
- Er muss 63 Byte durchgehend frei nutzbaren Platz bieten, denn wir dürfen unsere Sprite-Daten natürlich nicht in einen Speicherbereich schreiben, der bereits für andere Daten genutzt wird. 63 Byte deswegen, weil das 64. Byte nur als Platzhalter zum nächsten Block dient und nicht in die Spritedaten einfließt.

Durch 64 teilbare Adressen gibt es ja viele, aber wir müssen in dem Speicherbereich auch alle unsere 63 Bytes unterbringen können, ohne dabei andere Daten zu überschreiben.

Es hilft uns nichts, wenn die Adresse durch 64 teilbar ist, wir aber nur vielleicht 15 Bytes nutzen können, weil ab dem 16. Byte vielleicht bereits andere Daten folgen, die nicht überschrieben werden dürfen.

Glücklicherweise gibt es einige solcher frei verfügbaren Bereiche, welche diese Kriterien erfüllen und die wir daher zur Ablage unserer Sprite-Daten nutzen können.

Doch alles schön der Reihe nach.

Warum muss der Speicherbereich an einer durch 64 teilbaren Adresse beginnen?

Der Grund ist folgender:

Die 8 Speicherstellen von 2040 bis 2047 haben in Bezug auf Sprites eine wichtige Bedeutung.

Jede dieser 8 Speicherstellen ist einem Sprite zugeordnet, Speicherstelle 2040 ist Sprite 0 zugeordnet, Speicherstelle 2041 ist Sprite 1 zugeordnet, bis hin zur Speicherstelle 2047, welche Sprite 7 zugeordnet ist.

Jede dieser Speicherstellen enthält eine Blocknummer zwischen 0 und 255.

Diese Blocknummer multipliziert mit 64 ergibt dann jene Speicheradresse, die den Beginn des Speicherbereichs darstellt, in welchem wir die 63 Bytes Daten für unser Sprite ablegen.

Spielen wir das mal anhand der Speicherstelle 2040 durch, d.h. mit jener Speicherstelle, welche die Blocknummer für die Daten von Sprite 0 enthält.

Angenommen, sie enthielte die Blocknummer 0, dann würden die Spritedaten an Adresse $0 * 64 = 0$ beginnen. Diesen Block können wir jedoch nicht benutzen, denn wenn wir auf der Seite <https://www.c64-wiki.de/wiki/Zeropage> einen Blick auf die Belegung der Zeropage werfen, dann sehen wir, dass der Bereich von Adresse 0-63 bereits von anderen wichtigen Daten genutzt wird.

Probieren wir es mit Blocknummer 1, das wären dann die Adressen ab Adresse $1 * 64$, also Adresse 64. Tja, laut den Informationen auf der oben genannten Seite ist Block 1 leider auch schon vergeben.

Das geht leider weiter bis inklusive Block 10, also den Adressen 640 – 703.

Den Bereich mit der Blocknummer 11, also der Bereich von Adresse 704 bis 767, können wir jedoch für die Ablage unserer Spritedaten nutzen, da er nicht benutzt wird.

\$2C0 - \$2FF	704 - 767		Platz für Spritedatenblock 11, da nicht genutzt
---------------	-----------	--	---

Um es gleich vorweg zu nehmen:

Auch die Blöcke mit den Nummern 13, 14 und 15 können wir für unsere Spritedaten nutzen.

\$340 - \$37F	832 - 895		Platz für Spritedatenblock 13 (nur bei Nichtnutzung des Datasetten-/Kassettenpuffers!)
\$380 - \$3BF	896 - 959		Platz für Spritedatenblock 14 (nur bei Nichtnutzung des Datasetten-/Kassettenpuffers!)
\$3C0 - \$3FF	960 - 1023		Platz für Spritedatenblock 15 (nur bei Nichtnutzung des Datasetten-/Kassettenpuffers!)

Es gibt noch einiges anzumerken in Bezug auf die Blocknummern, doch das würde an dieser Stelle nur verwirren. Am Ende des Kapitels werde ich dies nachholen.

Festlegen der Blocknummer für die Spritedaten

Gut, dann nehmen wir doch für die Daten unseres Sprites gleich den ersten Block, den wir gefunden haben, also den mit der Nummer 11.

Wir fügen also folgende Zeile hinzu:

```
10 POKE 2040,11
```

Dadurch weiß der C64, dass die Daten für das Sprite 0 in Block 11 liegen, also ab der Speicheradresse 704 ($11 * 64$) zu finden sind.

Doch das ist erst die halbe Miete, denn bis jetzt stehen unsere Spritedaten nur in den DATA-Zeilen und noch nicht in dem Speicherblock 11.

Das Kopieren führen wir mittels folgender Schleife durch:

```
20 FOR I=0 TO 62  
30 READ A  
40 POKE 704+I,A  
50 NEXT I
```

Nun müssen wir noch eine ganze Reihe bestimmter Speicherstellen verändern, damit unser Sprite in der gewünschten Form auf dem Bildschirm angezeigt wird.

Typ des Sprites festlegen (einfarbig oder mehrfarbig)

In der Speicherstelle 53276 ist jedes der 8 Bits mit einem Sprite verbunden, Bit 0 mit Sprite 0 bis hin zu Bit 7, welches mit Sprite 7 verbunden ist. Setzt man ein Bit auf den Wert 0, dann gibt man dadurch an, dass es sich bei dem Sprite, welches mit diesem Bit verbunden ist, um ein einfarbiges Sprite handelt. Setzt man den Wert hingegen auf den Wert 1, dann gibt man dadurch an, dass es sich um ein mehrfarbiges Sprite handelt.

Da wir uns aktuell mit den einfarbigen Sprites beschäftigen, setzen wir das Bit an der Position 0 auf den Wert 0. Dadurch wird das Sprite 0 als einfarbig markiert.

Hier kommt uns nun unser Wissen über logische Verknüpfungen entgegen, denn wir müssen hier das Bit 0 auf den Wert 0 setzen.

Dazu brauchen wir folgende UND-Verknüpfung:

```
60 POKE 53276,PEEK(53276) AND (NOT (1))
```

Farbe des Sprites festlegen

Die Speicherstellen von 53287 bis 53294 enthalten die Farben für die 8 Sprites (falls es sich um einfarbige Sprites handelt)

Wir wählen für Sprite 0 die Farbe Weiß, also müssen wir den Wert 1 in die Speicherstelle 53287 schreiben.

```
70 POKE 53287,1
```

Festlegen der Spriteposition

Die Position eines Sprites wird durch eine Pixelposition in horizontaler und durch eine Pixelposition in vertikaler Richtung angegeben. In horizontaler Richtung (X) sind Werte von 0 bis

511 möglich und in vertikaler Richtung (Y) sind es Werte zwischen 0 und 255, wobei die Position X=0, Y=0 in der linken oberen Ecke des Bildschirms liegt.

Hier ist jedoch wirklich die linke obere Ecke des gesamten Bildschirms inklusive Rahmen gemeint, nicht die linke, obere Ecke des Ausgabebereichs, in dem beispielsweise die Textausgaben erfolgen.

Für die X-Koordinaten der 8 Sprites sind die Speicherstellen 53248, 53250, 53252, 53254, 53256, 53258, 53260 und 53262 zuständig, im Falle von Sprite 0 müssen wir die X-Koordinate also in der Speicherstelle 53248 ablegen.

Um das Sprite an den linken Rand des sichtbaren Bereichs zu positionieren, ist nicht, wie vielleicht vermutet, der Wert 0 erforderlich, sondern der Wert 24.

Die Einstellung der X-Koordinate führen wir mit dem Befehl

```
80 POKE 53248,24
```

durch.

Nun müssen wir uns noch um die Y-Koordinate kümmern.

Für die Y-Koordinaten der 8 Sprites sind die Speicherstellen 53249, 53251, 53253, 53255, 53257, 53259, 53261 und 53263 zuständig, im Falle von Sprite 0 müssen wir die Y-Koordinate also in der Speicherstelle 53249 ablegen.

Der Wert für den obersten Rand des sichtbaren Bereichs lautet 50.

Die Einstellung der Y-Koordinate für diese Position führen wir mit dem Befehl

```
90 POKE 53249,50
```

durch.

Festlegen der Sprite-Priorität in Bezug auf den Hintergrund

Dazu brauchen wir die Speicherstelle 53275. Auch diese Speicherstelle folgt dem bereits beschriebenen Schema und enthält für jedes Sprite ein eigenes Bit.

Enthält dieses Bit den Wert 0, dann hat das Sprite eine höhere Priorität als der Hintergrund und wird daher vor dem Hintergrund dargestellt. Enthält das jeweilige Bit jedoch den Wert 1, dann hat der Hintergrund höhere Priorität und das Sprite wird hinter dem Hintergrund dargestellt.

Wir entscheiden uns dafür, das Sprite vor dem Hintergrund darzustellen und setzen daher das Bit 0 auf den Wert 0.

```
100 POKE 53275,PEEK(53275) AND (NOT(1))
```

Sprite aktivieren

Nun müssen wir unser Sprite nur noch einschalten, damit es auch auf dem Bildschirm angezeigt wird.

In der Speicherstelle 53269 ist jedes der 8 Bits mit einem Sprite verbunden, Bit 0 mit Sprite 0 bis hin zu Bit 7, welches mit Sprite 7 verbunden ist. Setzt man ein Bit auf den Wert 1, dann wird das Sprite, das mit diesem Bit verbunden ist, angezeigt. Setzt man es umgekehrt auf den Wert 0, dann verschwindet das jeweilige Sprite.

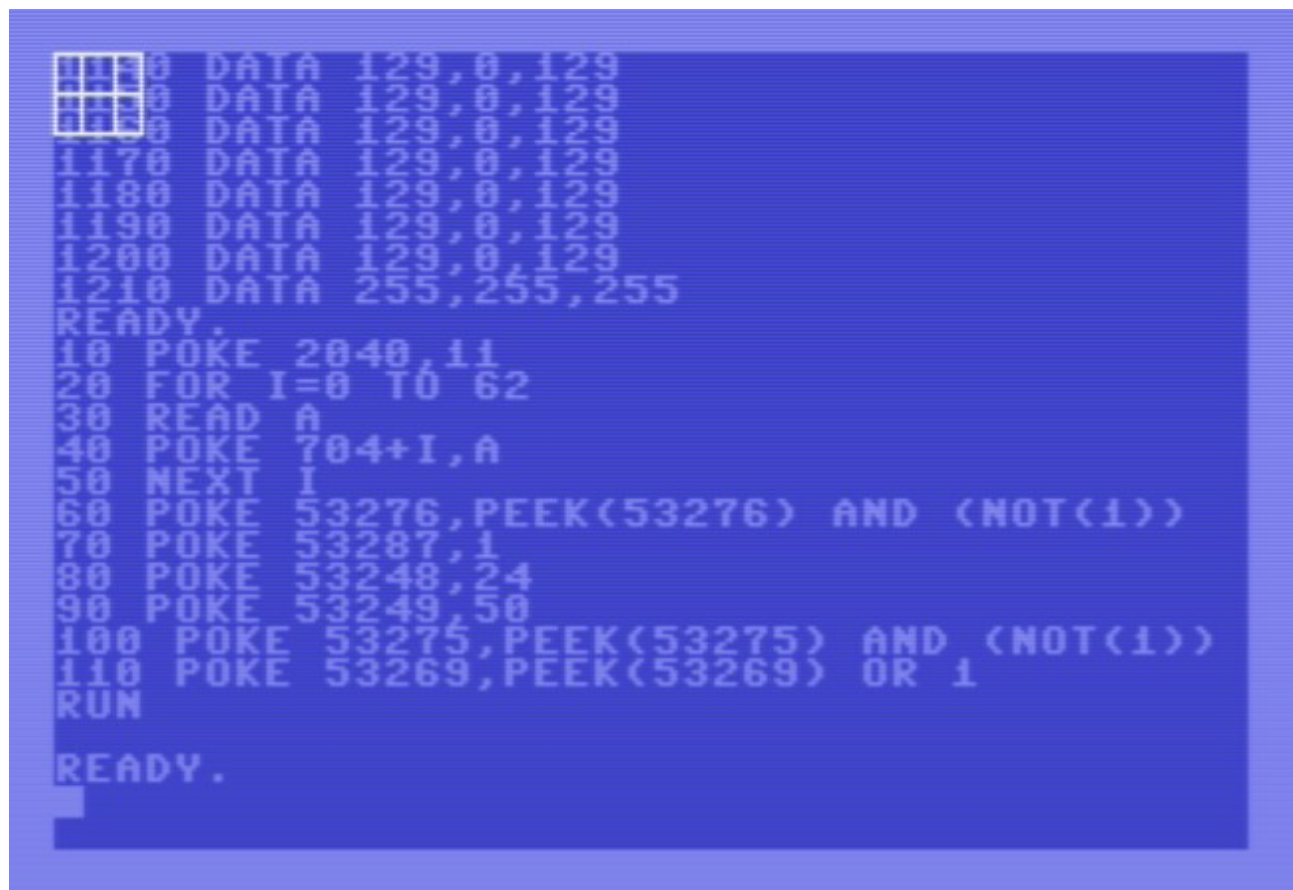
Wichtig:

Durch Aktivieren des Sprites wird das Sprite zwar grundsätzlich sichtbar gemacht, das bedeutet jedoch nicht, dass es sich gerade auch im sichtbaren Bereich auf dem Bildschirm befindet. Es kann je nach Koordinate beispielsweise vom Rahmen teilweise oder ganz verdeckt werden.

Setzen wir also Bit 0 in dieser Speicherstelle auf den Wert 1:

```
110 POKE 53269,PEEK(53269) OR 1
```

Wenn wir das Programm nun mit RUN starten, dann sollte folgendes zu sehen sein.



```
DATA 129,0,129
DATA 129,0,129
DATA 129,0,129
DATA 129,0,129
DATA 129,0,129
DATA 129,0,129
DATA 255,255,255
READY.
10 POKE 2040,11
20 FOR I=0 TO 62
30 READ A
40 POKE 704+I,A
50 NEXT I
60 POKE 53276,PEEK(53276) AND (NOT(1))
70 POKE 53287,1
80 POKE 53248,24
90 POKE 53249,50
100 POKE 53275,PEEK(53275) AND (NOT(1))
110 POKE 53269,PEEK(53269) OR 1
RUN
READY.
```

Es hat also soweit alles funktioniert und wir haben unser erstes Sprite auf dem Bildschirm dargestellt.

Wählen wir doch mal eine andere Farbe, z.B. Gelb (Farbcode 7) und geben gleich im Direktmodus den Befehl

```
POKE 53287,7
```

ein.

Das Sprite sollte nun in gelb angezeigt werden.

Lassen wir unser Sprite mal verschwinden? Aber sicher, das funktioniert mit dem Befehl

```
POKE 53269,PEEK(53269) AND (NOT(1))
```

Das Sprite sollte nun verschwunden sein.

Sichtbar machen können wir es wieder mit dem Befehl

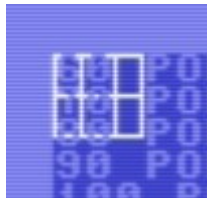
```
POKE 53269,PEEK(53269) OR 1
```

Das Sprite sollte nun wieder zu sehen sein.

Legen wir es doch mal hinter den Hintergrund, dazu ist der Befehl

```
POKE 53275,PEEK(53275) OR 1
```

nötig.



Nun befinden sich die BASIC-Zeilennummern im Vordergrund und überdecken das Sprite an manchen Stellen.

Hier zum Vergleich die vorherige Anzeige, bei der das Sprite im Vordergrund liegt und die Zeilennummern an manchen Stellen verdeckt.



Experimentieren wir nun ein wenig mit der Position des Sprites.

Verändern wir doch mal die X-Koordinate auf den Wert 100, was über den Befehl

```
POKE 53248,100
```

möglich ist.



Wichtig:

Wenn wir für unser Sprite eine X-Koordinate größer als 255 wählen, dann müssen wir hier einen anderen Weg einschlagen, denn in einem Byte kann man ja nur Werte zwischen 0 und 255 ablegen.

Hier sehen Sie die Position bei einer X-Koordinate von 255, also die höchstmögliche X-Koordinate, die in der Speicherstelle 53248 möglich ist.

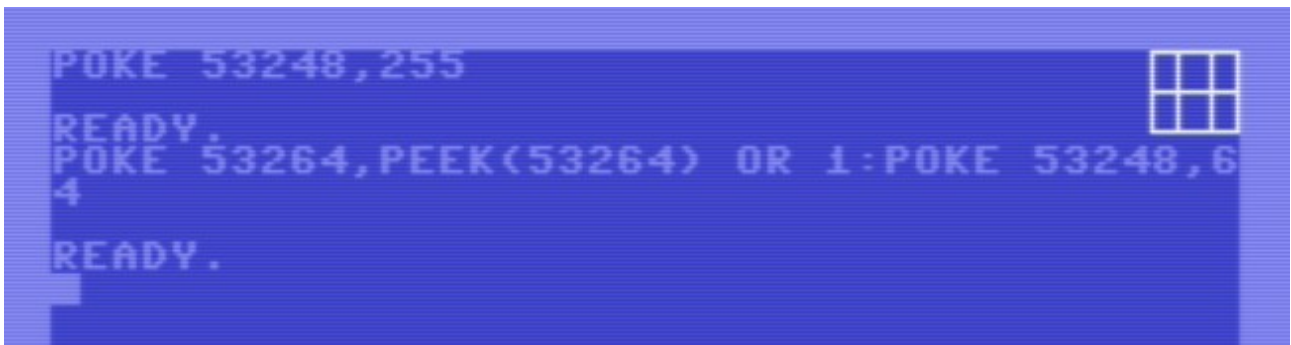


Wollen wir das Sprite an den rechten Rand des sichtbaren Bereichs positionieren, also auf die Position 320, dann müssen wir die Speicherstelle 53264 zu Hilfe nehmen.

Auch diese Speicherstelle enthält nach dem bereits erwähnten Schema für jedes Sprite ein eigenes Bit. Dieses Bit dient als zusätzliches Bit für die Darstellung von X-Koordinaten, welche größer als 255 sind und hat in Bezug auf die X-Koordinate die Wertigkeit 256.

Wir wollen das Sprite auf die X-Koordinate 320 setzen, d.h. wir müssen das Bit 0 (für das Sprite 0) in der Speicherstelle 53264 auf den Wert 1 setzen und den Rest, also was vom Wert 256 noch auf den Wert 320 fehlt, schreiben wir wie gehabt in die Speicherstelle 53248.

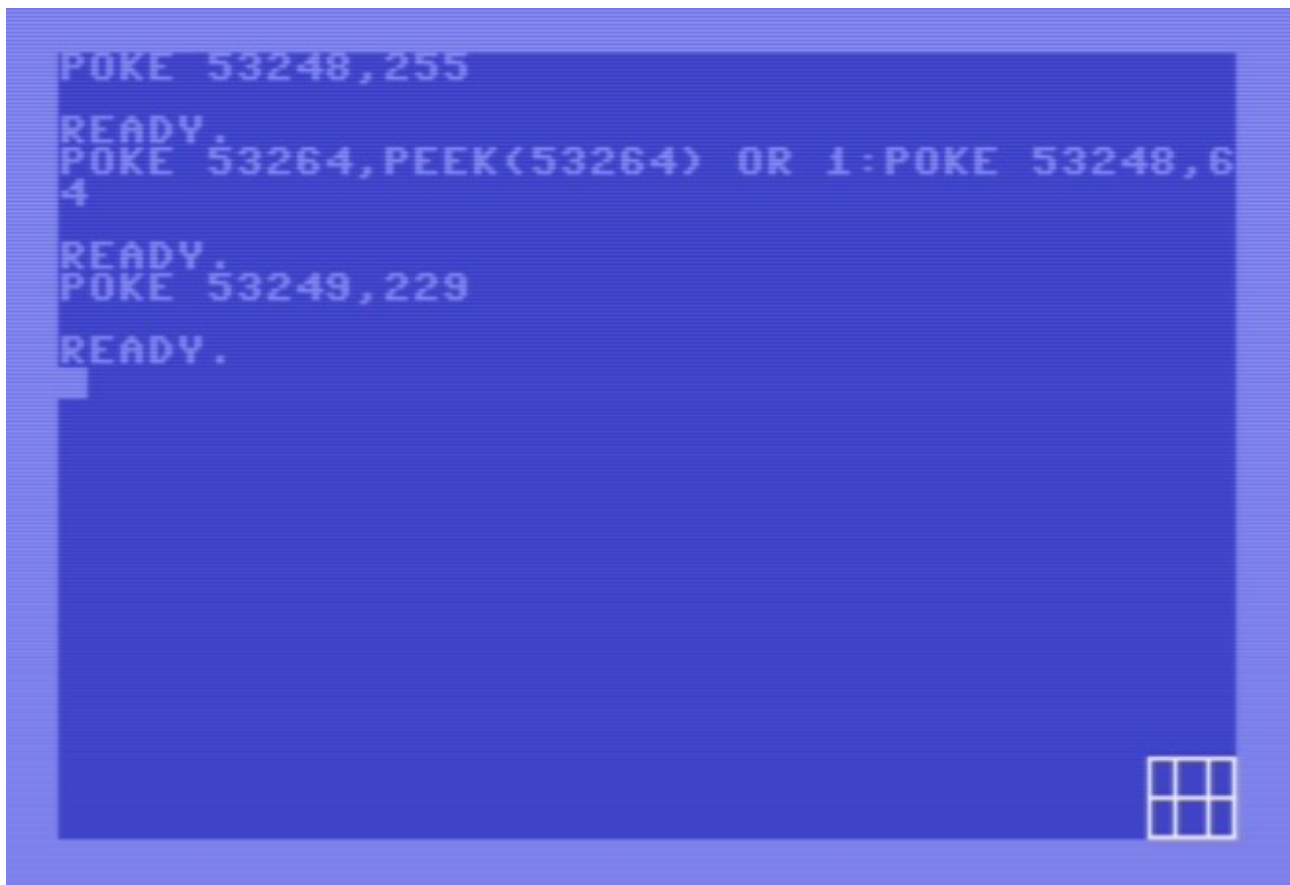
```
POKE 53264,PEEK(53264) OR 1:POKE 53248,64
```



Wichtig ist hier, dass wir das Bit 0 in Speicherstelle 53264 wieder auf 0 setzen, wenn wir die X-Koordinate auf einen Wert zwischen 0 und 255 setzen wollen.

Dies funktioniert mit dem Befehl `POKE 53264,PEEK(53264) AND (NOT(1))`

Nun verschieben wir noch mit dem Befehl `POKE 53249,229` das Sprite an den unteren Rand des sichtbaren Bildschirmbereichs.



Wir haben auch die Möglichkeit, das Sprite sowohl in horizontaler als auch in vertikaler Richtung zu vergrößern. Die Auflösung wird dadurch nicht verdoppelt, das Sprite wird nur doppelt so breit oder hoch dargestellt.

Für eine horizontale Vergrößerung ist die Speicherstelle 53277 zuständig. Auch hier ist jedes Sprite mit einem eigenen Bit vertreten. Setzt man es auf den Wert 1, so wird das Sprite in horizontaler Richtung verdoppelt. Setzt man es umgekehrt auf den Wert 0, so wird das Sprite in Normalgröße angezeigt.

Hier eine Vergrößerung des Sprites in horizontaler Richtung:



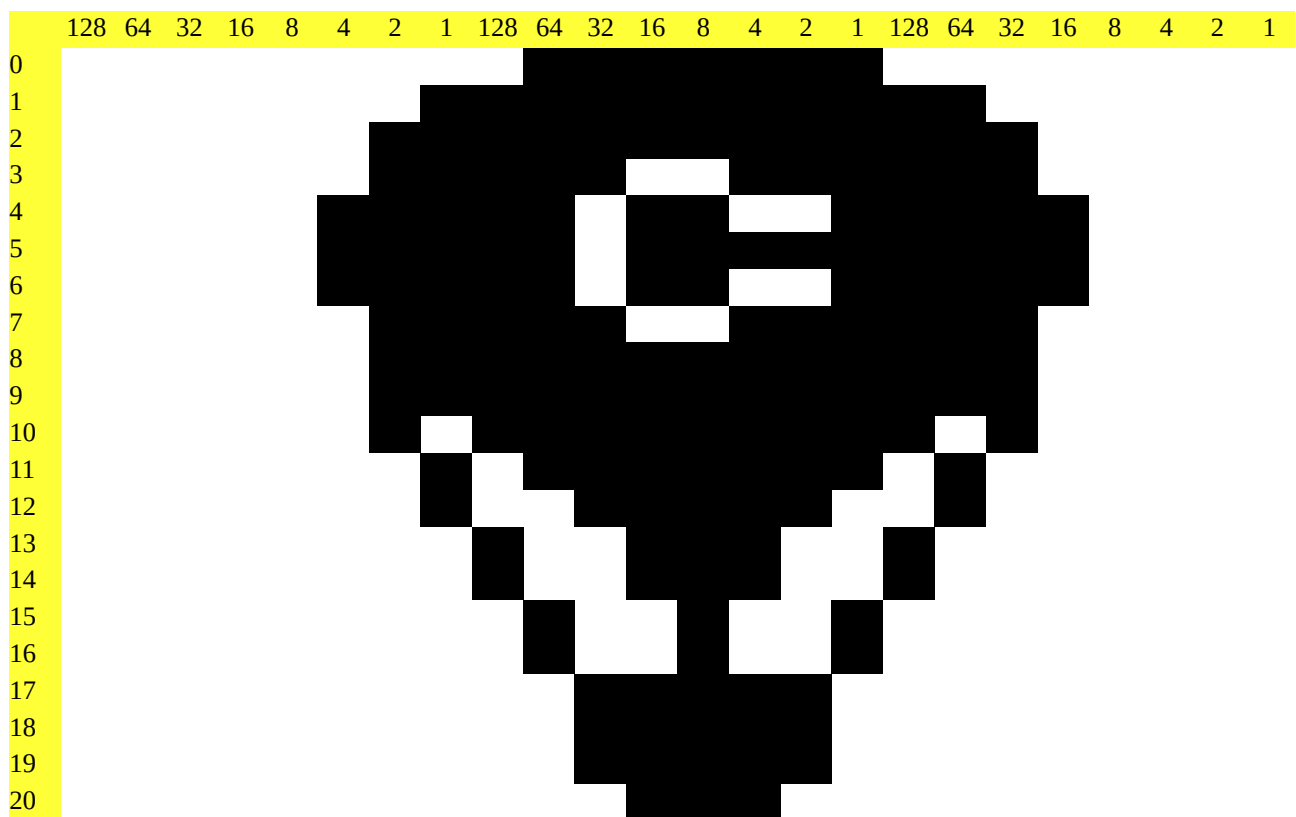
Für eine vertikale Vergrößerung ist die Speicherstelle 53271 zuständig. Auch hier ist wieder jedes Sprite mit einem eigenen Bit vertreten. Setzt man es auf den Wert 1, so wird das Sprite in vertikaler Richtung verdoppelt. Setzt man es umgekehrt auf den Wert 0, so wird das Sprite in Normalgröße angezeigt.

Hier eine Vergrößerung des Sprites in vertikaler Richtung:



Fügen wir nun ein weiteres Sprite hinzu.

Aus dem Handbuch des C64 kennen Sie sicherlich diesen Ballon, der dort als Beispiel für die Sprite-Programmierung verwendet wird. Bauen wir uns diesen doch mal nach.



Damit Sie sich nicht die Mühe machen brauchen, habe ich hier gleich die Tabelle mit den entsprechenden Bytes für Sie.

Erstes Byte	Zweites Byte	Drittes Byte
0	127	0
1	255	192
3	255	224
3	231	224
7	217	240
7	223	240
7	217	240
3	231	224
3	255	224
3	255	224
2	255	160
1	127	64
1	62	64
0	156	128
0	156	128
0	73	0
0	73	0
0	62	0
0	62	0
0	62	0
0	28	0

Hinweis:

Das Programm befindet sich unter dem Namen SPRITE2BAS auf der Diskette falls Sie es nicht selber abtippen möchten.

Wie bei unserem ersten Sprite legen wir diese Daten wieder in DATA-Zeilen ab.

```
1210 DATA 255,255,255
READY.
1220 REM DATEN FÜR SPRITE 1
1230 DATA 0,127,0
1240 DATA 1,255,192
1250 DATA 3,255,224
1260 DATA 3,231,224
1270 DATA 7,217,240
1280 DATA 7,223,240
1290 DATA 7,217,240
1300 DATA 3,231,224
1310 DATA 3,255,224
1320 DATA 3,255,224
1330 DATA 2,255,160
1340 DATA 1,127,64
1350 DATA 1,62,64
1360 DATA 0,156,128
1370 DATA 0,156,128
1380 DATA 0,73,0
1390 DATA 0,73,0
1400 DATA 0,62,0
1410 DATA 0,62,0
1420 DATA 0,62,0
1430 DATA 0,28,0
```

Dann führen wir exakt dieselben Schritte durch, die wir auch beim ersten Sprite durchgeführt haben.

Als Speicherort werden wir für unser zweites Sprite den Block Nummer 13 verwenden, denn wie bereits erwähnt, stehen die Blöcke 13, 14 und 15 zur freien Verfügung.

Wir ergänzen also folgende Zeile:

```
120 POKE 2041,13
```

Da wir dieses mal ja Sprite 1 meinen, müssen wir hier die Speicherstelle 2041 verwenden.

Nun kopieren wir die Spritedaten in den Block Nummer 13, dieser hat die Startadresse 832.

```
130 FOR I=0 TO 62
140 READ A
150 POKE 832+I,A
160 NEXT I
```

Typ des Sprites festlegen (einfarbig oder mehrfarbig)

Da es sich wieder um ein einfarbigen Sprite handelt, setzen wir das Bit an der Position 1 auf den Wert 0. Dadurch wird das Sprite 1 als einfarbig markiert.

Dazu brauchen wir folgende UND-Verknüpfung:

170 POKE 53276,PEEK(53276) AND (NOT (2))

Farbe des Sprites festlegen

Wir wählen für Sprite 1 die Farbe Gelb, also müssen wir den Wert 7 in die Speicherstelle 53288 schreiben.

180 POKE 53288,7

Festlegen der Spriteposition

Nehmen wir für dieses Sprite die X-Koordinate 160 und die Y-Koordinate 140.

190 POKE 53250,160

200 POKE 53251,140

Festlegen der Sprite-Priorität in Bezug auf den Hintergrund

Wir entscheiden uns auch bei diesem Sprite dafür, dass das Sprite vor dem Hintergrund dargestellt wird und wählen daher den Wert 0 für das Bit 1 in der Speicherstelle 53275.

210 POKE 53275,PEEK(53275) AND (NOT(2))

Sprite aktivieren

Setzen wir also Bit 1 in der Speicherstelle 53269 auf den Wert 1.

220 POKE 53269,PEEK(53269) OR 2

Nun starten wir das Programm mit RUN und es wird nun auch das zweite Sprite angezeigt.


```

20  FOR I=0 TO 62
30  READ A
40  POKE 704+I,A
50  NEXT I
60  POKE 53276,PEEK(53276) AND (NOT(1))
70  POKE 53287,1
80  POKE 53248,24
90  POKE 53249,50
100 POKE 53275,PEEK(53275) AND (NOT(1))
110 POKE 53269,PEEK(53269) OR 1
120 POKE 2041,13
130 FOR I=0 TO 62
140 READ A
150 POKE 832+I,A
160 NEXT I
170 POKE 53276,PEEK(53276) AND (NOT(2))
180 POKE 53288,7
190 POKE 53250,160
200 POKE 53251,140
210 POKE 53275,PEEK(53275) AND (NOT(2))
220 POKE 53269,PEEK(53269) OR 2
1000 REM DATEN FUER SPRITE 0
READY.

```

Nun bauen wir uns noch ein drittes Sprite, das Zeichnen ist dieses mal sehr einfach, da es die komplette Fläche ausfüllt.

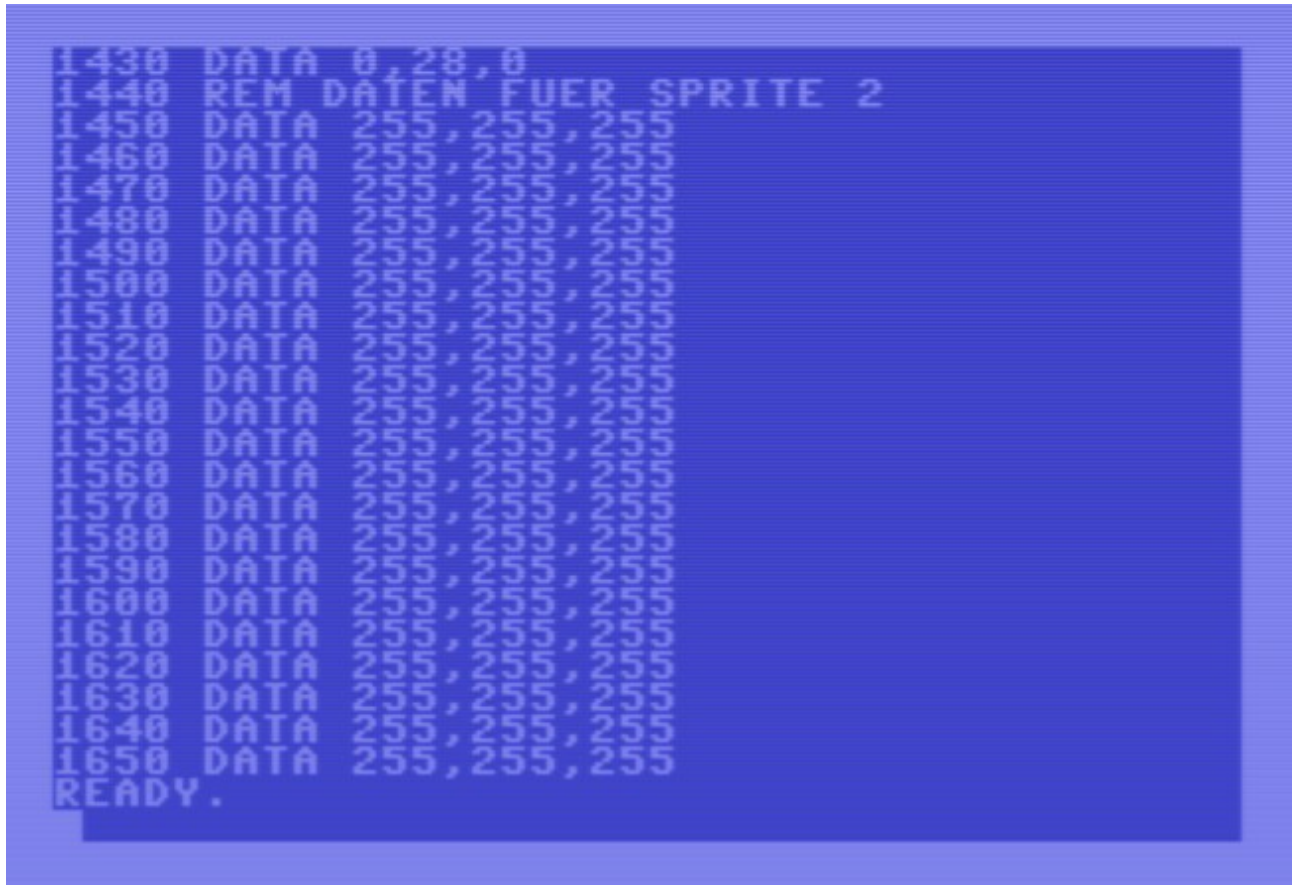
	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								

Das macht die Ermittlung der Werte für die DATA-Zeilen natürlich auch recht einfach.

Wir benötigen für die 21 DATA-Zeilen immer denselben Inhalt 255,255,255.

Hinweis:

Das Programm befindet sich unter dem Namen SPRITE3BAS auf der Diskette falls Sie es nicht selber abtippen möchten.



Dann ergänzen wir noch folgende Zeilen, um die Einstellungen für das Sprite 2 vorzunehmen. Als Speicherblock verwenden wir dieses mal den Block 14 (Startadresse 896).

Als Farbe wählen wir Türkis, für die X-Koordinate 100 und für die Y-Koordinate ebenfalls 100.

```
230 POKE 2042,14
240 FOR I=0 TO 62
250 READ A
260 POKE 896+I,A
270 NEXT I
280 POKE 53276,PEEK(53276) AND (NOT(4))
290 POKE 53289,3
300 POKE 53252,100
310 POKE 53253,100
320 POKE 53275,PEEK(53275) AND (NOT(4))
330 POKE 53269,PEEK(53269) OR 4
```

Wir starten das Programm wieder mit RUN und nun sehen wir auch unser drittes Sprite.

Wie auch bei dem Ballon sieht man bei diesem Sprite besonders gut, dass es vor dem Hintergrund liegt, weil es die Werte in den DATA-Zeilen verdeckt.



Sprite-Priorität

Ich habe über die entsprechenden POKE-Befehle Sprite 0 auf die Position 87,87 verschoben und Sprite 1 auf die Position 110,110 damit man die Prioritäten der Sprites erkennen kann.



Hier sieht man, dass Sprite 2 (Viereck) sowohl von Sprite 0 (Gitter) als auch von Sprite 1 (Ballon) überdeckt wird.

Die weißen Linien des Gitters überdecken die türkisen Stellen des Vierecks.

Das liegt daran, dass Sprite 0 die höchste Priorität hat. Das geht weiter bis Sprite 7 mit der niedrigsten Priorität.

Die weißen Linien des Gitters würden auch den Ballon stellenweise überdecken, weil die Priorität des Gitters höher ist. Je niedriger die Nummer des Sprites ist, desto höher ist die Priorität gegenüber den Sprites mit höheren Nummern.

Diese Prioritäten kann man nicht verändern. Sprite 0 wird also immer die höchste und Sprite 7 immer die niedrigste Priorität haben. Es ist also nicht möglich, beispielsweise Sprite 2 eine höhere Priorität als Sprite 1 zu geben.

Die Priorität der Sprites untereinander ist nicht zu verwechseln mit der Priorität, welche die einzelnen Sprites in Bezug auf den Hintergrund haben.

Auf dem Bild sieht man, dass alle Sprites den BASIC-Code verdecken, weil wir dies bei jedem Sprite über die Speicherstelle 53275 so eingestellt haben.

So, nun wollen wir das alles mal in Assembler umsetzen. Logische Verknüpfungen sind hier sehr stark vertreten. Wenn Sie diesbezüglich fit sind, sollten Sie keine Probleme damit haben, den Code nachvollziehen zu können.

Was die Positionierung der Sprites betrifft, ist jedoch etwas Mühe gefordert, da ich hier auch die Positionierung der Sprites an x-Koordinaten berücksichtige, welche größer als 255 sind. Ich habe die Schritte jedoch auf Unterprogramme aufgeteilt, damit man die Abläufe besser nachvollziehen kann.

Laden Sie dazu den Sourcecode spr1asm in den Editor, sodass wir ihn gemeinsam Schritt für Schritt durchgehen können.

Folgende Schritte werden im Programm durchlaufen:

Vergeben der Blocknummern für die drei Sprites

Speicherstelle \$07F8 (dezimal 2040) => Blocknummer \$0B (dezimal 11)

Speicherstelle \$07F9 (dezimal 2041) => Blocknummer \$0D (dezimal 13)

Speicherstelle \$07FA (dezimal 2042) => Blocknummer \$0E (dezimal 14)

```
    ; gitter  = sprite 0, block 11
    ; ballon  = sprite 1, block 12
    ; viereck = sprite 2, block 13

    ; blocknr. fuer gitter
    lda #$0b
    sta $07f8

    ; blocknr. fuer ballon
    lda #$0d
    sta $07f9

    ; blocknr. fuer viereck
    lda #$0e
    sta $07fa
```

Umkopieren der Spritedaten

Dies können wir für alle drei Sprites in einer Schleife erledigen. Die Felder mit den Spritedaten (spritegitter, spriteballon und spriteviereck) finden Sie am Ende des Programms.

```
    ; spritedaten kopieren
    ; spritegitter -> block 11
    ; spriteballon -> block 13
    ; spriteviereck -> block 14

copyloop    lda #$00
            lda spritegitter,x
            sta $02c0,x

            lda spriteballon,x
            sta $0340,x

            lda spriteviereck,x
            sta $0380,x

            inx
```

```
cpx #$3f
bne copyloop
```

Der Inhalt des X Registers wird wie gewohnt innerhalb der Schleife von 0 beginnend hochgezählt, bis es den Wert \$3F (dezimal 63) enthält. Solange dies nicht der Fall ist, wird immer wieder zum Label copyloop gesprungen und die Schleife erneut durchlaufen. Auf diese Weise wird Byte für Byte in den jeweiligen Block kopiert.

Sprites einschalten

```
; sprite 0,1,2 einschalten
lda $d015
ora #$07
sta $d015
```

Dazu müssen wir in der Speicherstelle \$D015 (dezimal 53269) die Bits 0, 1 und 2 setzen. Als Erstes lesen wir den Inhalt der Speicherstelle \$D015 aus, führen mit diesem eine ODER-Verknüpfung mit dem Wert 7 (binär %00000111) durch, wodurch die Bits 0, 1 und 2 gesetzt werden. Abschließend schreiben wir den Wert wieder in die Speicherstelle \$D015 zurück.

Sprites als einfarbig definieren

```
; alle sprites einfarbig
lda $d01c
and $f8
sta $d01c
```

Dazu müssen wir in der Speicherstelle \$D01C (dezimal 53276) die Bits 0, 1 und 2 zurücksetzen. Als Erstes lesen wir den Inhalt der Speicherstelle \$D01C aus, führen mit diesem eine UND-Verknüpfung mit dem Wert \$F8 (binär %11111000) durch, wodurch die Bits 0, 1 und 2 zurückgesetzt werden. Abschließend schreiben wir den Wert wieder in die Speicherstelle \$D01C zurück.

Farben für die Sprites vergeben

```
; farbe fuer gitter
lda #$01
sta $d027

; farbe fuer ballon
lda #$07
sta $d028

; farbe fuer viereck
lda #$03
sta $d029
```

Das Gitter (Sprite 0) erhält die Farbe Weiß, d.h. wir müssen den Wert 1 in die Speicherstelle \$D027 (dezimal 53287) schreiben.

Der Ballon (Sprite 1) soll in gelber Farbe angezeigt werden, was durch den Wert 7 in Speicherstelle \$D028 (dezimal 53288) erreicht wird.

Das Viereck (Sprite 2) wollen wir in Türkis anzeigen und schreiben dazu den Wert 3 in die Speicherstelle \$D029 (dezimal 53289).

Priorität der Sprites gegenüber dem Hintergrund einstellen

```
; alle sprites vor hintergrund
lda $d01b
and $f8
sta $d01b
```

Alle unsere Sprites sollen Priorität gegenüber dem Hintergrund haben, d.h. wir müssen die Bits 0, 1 und 2 in der Speicherstelle \$D01B (dezimal 53275) zurücksetzen. Als Erstes lesen wir den Inhalt der Speicherstelle \$D01B aus, führen mit diesem eine UND-Verknüpfung mit dem Wert \$F8 (binär %11111000) durch, wodurch die Bits 0, 1 und 2 zurückgesetzt werden. Abschließend schreiben wir den Wert wieder in die Speicherstelle \$D01B zurück.

So, nun kommen wir zu dem Thema, in das nun etwas Mühe investiert werden muss. Das Gute daran ist jedoch, dass Sie dann den schwierigsten Teil hinter sich haben und Ihnen die nächsten Programmbeispiele keine großen Schwierigkeiten mehr bereiten sollten.

Positionieren der Sprites

Wie wir bereits wissen, sind in den Speicherstellen \$D000, \$D002, \$D004, \$D006, \$D008, \$D00A, \$D00C und \$D00E die x-Koordinaten der Sprites gespeichert.

In diesen Speicherstellen können wir jedoch nur Werte zwischen 0 und 255 speichern. Wie man x-Koordinaten größer als 255 einstellt und welche Rolle die Speicherstelle \$D010 (dezimal 53264) dabei spielt, habe ich bereits bei der BASIC-Umsetzung erklärt. Trotzdem möchte ich die Zusammenhänge noch einmal wiederholen, um sie wieder in Erinnerung zu rufen.

Um auch die x-Koordinaten jenseits der Grenze von 255 nutzen zu können, gibt es die Speicherstelle \$D010. Sie stellt für jedes der 8 Sprites ein zusätzliches Bit für die Festlegung der x-Koordinate zur Verfügung, wodurch die vorhin genannten Speicherstellen quasi um ein zusätzliches Bit erweitert werden. Dieses zusätzliche Bit reicht aus, um auch alle möglichen x-Koordinaten von 256 bis 511 darstellen zu können.

Angenommen, wir wollen für das Sprite 0 eine x-Koordinate von 320 einstellen. Mit der Speicherstelle \$D000 allein kommen wir hier nicht aus, da wir dort ja nur Werte zwischen 0 und 255 speichern können. Wir müssen also das Bit 0 in der Speicherstelle \$D010 zu Hilfe nehmen, um die x-Koordinate 320 einstellen zu können.

Bit 0 aus \$D010	Speicherstelle \$D000							
8	7	6	5	4	3	2	1	0
256	128	64	32	16	8	4	2	1
1	0	1	0	0	0	0	0	0

Das zusätzliche Bit hat die Wertigkeit 256, d.h. wir müssen in der Speicherstelle \$D000 nur mehr den Wert eintragen, der uns noch auf den Wert 320 fehlt.

$320 - 256 = 64$, d.h. um die x-Koordinate für das Sprite 0 auf den Wert 320 zu setzen, müssen wir Bit 0 in der Speicherstelle \$D010 setzen und den Wert 64 in die Speicherstelle \$D000 schreiben.

Dieses Schema gilt analog auch für die anderen Sprites.

Kommen wir nun zu den drei Unterprogrammen, welche an der Positionierung der Sprites beteiligt sind.

Unterprogramm setbit8forx

```
; unterprogramm setbit8forx
;
; setzt oder loescht fuer ein sprite
; das bit8 fuer die x-koordinate
; in der speicherstelle $d010
;
; parameter:
; akku = spritenummer 0..7
; carryflag = 0: bit loeschen
; carryflag = 1: bit setzen

setbit8forx
    tay
    lda zweierpotenzen,y
    bcc clearbit
    ora $d010
    jmp setd010
clearbit
    eor #$ff
    and $d010
setd010
    sta $d010
    rts
```

Das Unterprogramm übernimmt zwei Parameter. Im Y Register wird die Nummer des Sprites übergeben, dessen Bit man in der Speicherstelle \$D010 ändern will. Will man also das Bit für das Sprite 0 ändern, dann muss man im Y Register den Wert 0 übergeben. Für das Sprite 7 wäre es der Wert 7.

Der zweite Parameter kommt über das Carry Flag ins Unterprogramm. Setzt man es vor dem Aufruf des Unterprogramms, dann bedeutet das, dass man das jeweilige Bit setzen will. Ist es hingegen zurückgesetzt, dann signalisiert man dadurch, dass man das jeweilige Bit zurücksetzen will.

Im Datenbereich ist ein Feld namens zweierpotenzen definiert. Hier stehen nacheinander die Wertigkeiten der Bitpositionen in einem Byte, also die Werte 1, 2, 4, 8, 16, 32, 64, 128 (im Assemblercode habe ich sie jedoch hexadezimal angegeben: \$01, \$02, \$04, \$08, \$10, \$20, \$40, \$80)

```
zweierpotenzen
    .byte $01,$02,$04,$08
    .byte $10,$20,$40,$80
```

Durch den ersten Befehl TAY wird die Nummer des Sprites in das Y Register kopiert, damit wir über die indizierte Adressierung auf den Bereich zweierpotenzen zugreifen können.

Wozu brauchen wir diese Werte? Wir haben als Parameter eine Spritenummer übergeben. Diese Nummer entspricht 1:1 der Position des Bits in der Speicherstelle \$D010, welches für das jeweilige Sprite zuständig ist.

Bitposition 0 ist für das Sprite 0 zuständig, Bitposition 1 ist für das Sprite 1 zuständig usw. Durch die Spritenummer kennen wir also gleichzeitig die Position des Bits in der Speicherstelle \$D010, welches wir ändern müssen.

Durch den Befehl LDA zweierpotenzen,y wird die Wertigkeit an dieser Bitposition in den Akkumulator geladen. Im Falle von Sprite 0 also der Wert 1, im Falle von Sprite 1 der Wert 2 usw. Diesen Wert brauchen wir für die anschliessende logische Verknüpfung.

Falls das Carry Flag nicht gesetzt ist, wird durch den Befehl BCC (branch on carry clear) zum Label clearbit verzweigt. Dort wird durch den Befehl EOR #\$FF zunächst das Einerkomplement des Akkumulator-Inhalts gebildet und das Ergebnis anschließend über den Befehl AND \$D010 eine UND-Verknüpfung mit dem aktuellen Inhalt der Speicherstelle \$D010 durchgeführt. Dies bewirkt ein Zurücksetzen des jeweiligen Bits im Akkumulator-Inhalt.

Im Anschluss wird das Ergebnis durch den Befehl STA \$D010 in die Speicherstelle \$D010 zurückgeschrieben, damit die Änderung wirksam wird. Durch den Befehl RTS erfolgt dann der Rücksprung aus dem Unterprogramm.

Ist das Carry Flag hingegen gesetzt, wird über den Befehl ORA \$D010 eine ODER-Verknüpfung des Akkumulator-Inhalts mit dem aktuellen Inhalt der Speicherstelle \$D010 durchgeführt, wodurch das jeweilige Bit im Akkumulator-Inhalt gesetzt wird.

Dann wird zum Label setd010 gesprungen, wodurch analog zum anderen Fall das Ergebnis in die Speicherstelle \$D010 zurückgeschrieben wird und über den Befehl RTS der Rücksprung aus dem Unterprogramm erfolgt.

Unterprogramm setspritex

```
; unterprogramm setspritex
; setzt die x-koordinate fuer ein sprite
;
; parameter:
; akku = spritenummer 0..7
; speicherstelle $fa = lo(x)
; speicherstelle $fb = hi(x)

setspritex
    pha
    asl a
    tay
    lda $fa
    sta $d000,y
    lda $fb
    beq xklg255
    pla
    sec
    jsr setbit8forx
    jmp setxende
xklg255
    pla
    clc
    jsr setbit8forx
setxende
    rts
```

Dieses Unterprogramm ist für die Einstellung der x-Koordinate eines Sprites zuständig und nutzt dafür das soeben beschriebene Unterprogramm setbit8forx.

Als Parameter wird im Akkumulator wieder die Nummer des Sprites übergeben, dessen x-Koordinate man einstellen will. Die x-Koordinate selbst setzt sich aus dem Inhalt der Speicherstellen \$FA (niederwertiges Byte) und \$FB (höherwertiges Byte) zusammen.

Vor dem Aufruf des Unterprogramms muss also die Nummer des Sprites im Akkumulator und die gewünschte x-Koordinate aufgeteilt auf das niederwertige und höherwertige Byte in den Speicherstellen \$FA und \$FB stehen.

Mit dem ersten Befehl PHA wird die übergebene Spritenummer auf dem Stack gesichert, da sie gleich durch den nächsten Befehl verändert wird.

Dies ist der neue Befehl ASL (Arithmetic Shift Left)

Dieser Befehl bewirkt, dass alle Bits im Akkumulator um eine Position nach links geschoben werden, wobei das Bit 7, welches an der linken Stelle dadurch herausfällt, in das Carry Flag wandert. Auf der rechten Seite kommt ein 0-Bit herein.

Hier ein Beispiel:

Angenommen der Akkumulator enthält den binären Wert %10110010:

Vor ASL	1	0	1	1	0	0	1	0
Nach ASL	0	1	1	0	0	1	0	0

Das Bit 7 mit dem Wert 1 ist herausgefallen und wird in das Carryflag übertragen, dieses wird also gesetzt.

Auf der rechten Seite kam ein 0-Bit herein.

Mathematisch gesehen bewirkt eine Verschiebung um eine Position nach links einer Multiplikation mit zwei.

Umgekehrt bewirkt eine Verschiebung um eine Position nach rechts einer Division durch zwei. Hierzu gibt es den Befehl LSR (Logical Shift Right). Umgekehrt wandert hier auf der linken Seite ein 0-Bit herein und das rechte herausfallende Bit wird durch das Carryflag aufgefangen.

Vor LSR	1	0	1	1	0	0	1	0
Nach LSR	0	1	0	1	1	0	0	1

Das Bit 0 mit dem Wert 0 ist herausgefallen und wird in das Carryflag übertragen, wodurch dieses zurückgesetzt wird.

Auf der linken Seite kam ein 0-Bit herein. Wir brauchen die Spritenummer später jedoch noch, daher müssen wir sie vor der Multiplikation auf dem Stack sichern.

Durch den Befehl TAY wird das Ergebnis der Multiplikation in das Y Register kopiert, damit wir über die indizierte Adressierung auf die Speicherstellen zugreifen können, welche für die x-Koordinate der Sprites zuständig sind.

Diese sind jeweils um zwei Stellen verschoben, weswegen es zuvor nötig war, die Spritenummer mit zwei zu multiplizieren.

Hier eine Tabelle, welche veranschaulicht, wie die Adresse der Speicherstellen für die jeweilige x-Koordinate durch Angabe der Spritenummer gebildet wird.

Spritenummer	Spritenummer * 2	Adresse
0	0	\$D000
1	2	\$D002
2	4	\$D004
3	6	\$D006
4	8	\$D008
5	10	\$D00A
6	12	\$D00C
7	14	\$D00E

Durch den Befehl STA \$D000,y wird das niederwertige Byte der x-Koordinate in diese Speicherstelle geschrieben. Dieses Byte wurde vorher durch den Befehl LDA \$FA in den Akkumulator geladen.

Kommen wir nun zum höherwertigen Byte der gewünschten x-Koordinate. Diese findet das Unterprogramm in der Speicherstelle \$FB vor und deshalb laden wir es mit dem Befehl LDA \$FB in den Akkumulator.

Falls das höherwertige Byte den Wert 0 enthält, es sich also um eine x-Koordinate kleiner oder gleich 255 handelt, dann wird zum Label xklg255 verzweigt. Dort wird mit dem Befehl PLA die zuvor auf dem Stack gesicherte Spritenummer wieder vom Stack in den Akkumulator geholt, denn diese brauchen wir nun für den Aufruf des Unterprogramms setbit8forx.

Da es sich um eine x-Koordinate kleiner oder gleich 255 handelt, wird mit dem Befehl CLC das Carry Flag zurückgesetzt und das Unterprogramm setbit8forx aufgerufen. Dadurch wird das jeweilige Bit in der Speicherstelle \$D010 zurückgesetzt. Durch den Befehl RTS erfolgt dann der Rücksprung aus dem Unterprogramm.

Enthält das höherwertige Byte jedoch einen Wert ungleich 0, handelt es sich also um eine x-Koordinate, die größer als 255 ist, dann wird ebenfalls zunächst die zuvor auf dem Stack gesicherte Spritenummer mit dem Befehl PLA in den Akkumulator geholt, da wir sie für den Aufruf des Unterprogramms setbit8forx brauchen. Da es sich um eine x-Koordinate größer als 255 handelt, muss das dem Sprite zugehörige Bit in der Speicherstelle \$D010 gesetzt werden.

Daher wird vor dem Aufruf des Unterprogramms setbit8forx das Carry Flag gesetzt, wodurch das soeben erwähnte Bit gesetzt wird. Nach der Rückkehr aus dem Unterprogramm wird zum Label setxende verzweigt. Dort erfolgt dann mittels des Befehls RTS der Rücksprung aus dem Unterprogramm.

Nun ist abhängig von der gewünschten x-Koordinate das niederwertige Byte in der korrekten Speicherstelle eingetragen und das jeweilige Bit in der Speicherstelle \$D010 entweder gesetzt oder nicht.

Unterprogramm setspritey

```
; unterprogramm setspritey
; setzt die y-koordinate fuer ein sprite
;
; parameter:
; akku = spritenummer 0..7
; speicherstelle $fa = y-koordinate

setspritey
    asl    a
    tay
    lda    $fa
    sta    $d001,y
    rts
```

Dieses Unterprogramm ist für die Einstellung der y-Koordinate eines Sprites zuständig. Hier haben wir es leichter als bei der x-Koordinate, weil hier keine Werte über 255 möglich sind.

Auch hier wird im Akkumulator die Spritenummer übergeben und die gewünschte y-Koordinate muss sich vor dem Aufruf des Unterprogramms in der Speicherstelle \$FA befinden.

Da die Speicherstellen für die y-Koordinaten ebenfalls jeweils um zwei Stellen versetzt sind, wird die Spritenummer wiederum durch den Befehl ASL mit zwei multipliziert und das Ergebnis in das Y Register kopiert, damit wir über die indizierte Adressierung auf die Speicherstellen zugreifen können, welche für die y-Koordinate der Sprites zuständig sind.

Hier wieder eine Tabelle, welche veranschaulicht, wie die Adresse der Speicherstellen für die jeweilige y-Koordinate durch Angabe der Spritenummer gebildet wird.

Spritenummer	Spritenummer * 2	Adresse
0	0	\$D001
1	2	\$D003
2	4	\$D005
3	6	\$D007
4	8	\$D009
5	10	\$D00B
6	12	\$D00D
7	14	\$D00F

Durch den Befehl STA \$FA wird dann die gewünschte y-Koordinate in diese Speicherstelle geschrieben. Zuvor haben wir die y-Koordinate mit dem Befehl LDA \$FA in den Akkumulator geladen. Und das war's auch schon, sodass mit dem Befehl RTS der Rücksprung aus dem Unterprogramm erfolgen kann.

Möglicherweise haben Sie sich gefragt, warum ich die Einstellung der x- und y-Koordinate auf separate Unterprogramme aufgeteilt habe. Ursprünglich hatte ich beide Einstellungen in einem Unterprogramm, aber es hat sich später herausgestellt, dass es besser ist, die beiden Einstellungen auf zwei separate Unterprogramme aufzuteilen.

Angenommen, man will ein Sprite horizontal über den Bildschirm bewegen. Dann verändert sich nur die x-Koordinate, aber die y-Koordinate bleibt konstant, sodass man sie auch nicht bei jedem Schritt immer wieder neu einstellen muss.

Dasselbe gilt für die vertikale Bewegung. Hier ändert sich nur die y-Koordinate und die x-Koordinate bleibt konstant. Gerade bei einer vertikalen Bewegung spart man hier einiges an Rechenzeit, weil das Einstellen der x-Koordinate ja doch ein wenig aufwändiger ist, als das Einstellen der y-Koordinate wie wir gesehen haben.

Durch die Aufteilung auf zwei Unterprogramme muss man vor dem Start der vertikalen Bewegung die gewünschte x-Koordinate nur ein einziges mal einstellen und nicht bei jedem Schritt in vertikaler Richtung.

Gleiches gilt natürlich auch für die horizontale Bewegung, hier stellt man die gewünschte y-Koordinate vor dem Start der horizontalen Bewegung ein einziges mal ein und erspart sich die Einstellung bei jedem Schritt in horizontaler Richtung.

Möchte man sowohl die x- als auch die y-Koordinate ändern, dann ruft man die beiden Unterprogramme setspritex und setspritey einfach hintereinander mit den gewünschten Werten auf.

Auf diese Art und Weise führt man die Schritte wirklich nur dann aus, wenn sie wirklich nötig sind. Kommen wir nun zu dem Programmteil, welcher die drei Sprites am Bildschirm positioniert und sich dabei der Unterprogramme bedient, die wir soeben besprochen haben.

Ich werde hier nur die Positionierung des Gitters erläutern, der Ballon und das Viereck werden auf dieselbe Art und Weise positioniert.

```
; gitter positionieren
; x = 320, y = 50

lda #$40 ; lo(x)
sta $fa ; in $fa
lda #$01 ; hi(x)
sta $fb ; in $fb
lda #$00 ; spritenummer 0
jsr setspritex
lda #$32 ; y
sta $fa ; in $fa
lda #$00 ; spritenummer 0
jsr setspritey
```

Die x-Koordinate des Gitters soll auf den Wert 320 eingestellt werden. Dies entspricht dem hexadezimalen Wert \$0140. Das niederwertige Byte \$40 schreiben wir in die Speicherstelle \$FA und das höherwertige Byte \$01 schreiben wir in die Speicherstelle \$FB.

Abschließend schreiben wir noch die Spritenummer 0 in den Akkumulator und rufen dann das Unterprogramm setspritex auf.

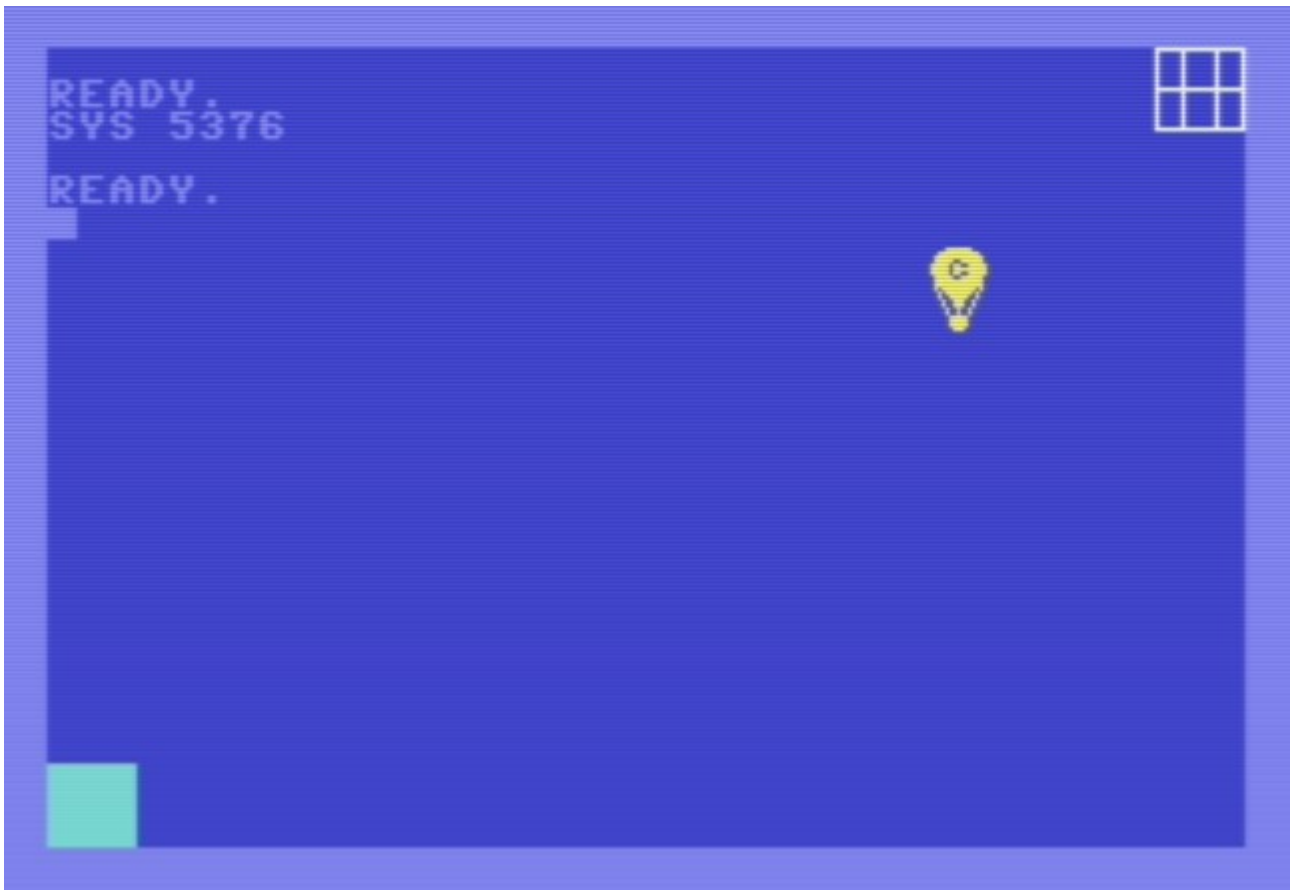
Dann schreiben wir die y-Koordinate in die Speicherstelle \$FA, laden den Akkumulator mit der Spritenummer 0 und rufen das Unterprogramm setspritey auf.

Somit haben wir das Gitter auf die Koordinaten x=320, y=50 positioniert.

Im Assemblercode folgt nun nur noch die Positionierung des Ballons und des Vierecks, welche wie bereits erwähnt vollkommen gleich funktioniert wie die des Gitters.

Den Abschluss des Programms bildet der Befehl RTS.

Wenn Sie das Programm mit SYS 5376 starten, sollten Sie folgendes Ergebnis erhalten:



Im nächsten Programm bringen wir wieder etwas Bewegung ins Spiel und wenden vieles von dem an, was wir bisher gelernt haben.

Was soll das Programm machen? Es soll einen gelben Ballon vom linken zum rechten Rand des Bildschirms bewegen. Bevor die Bewegung startet, soll auf einen Tastendruck gewartet werden.

Laden Sie die Datei spr2asm in den Editor, assemblieren den Assemblercode und starten das erzeugte Programm, damit Sie auch konkret sehen können, was sich da so tut. Zu Beginn wird der gelbe Ballon in der linken, oberen Ecke angezeigt und auf einen Tastendruck gewartet. Sobald dieser erfolgt ist, bewegt sich der Ballon vom linken zum rechten Rand des Bildschirms.

Wechseln Sie nun mit SYS 32768 zurück zum TMP, damit wir uns den Assemblercode zu Gemüte führen können.

Die meisten Unterprogramme kennen wir bereits aus dem vorherigen Beispiel, deswegen werden wir uns nur jene Unterprogramme ansehen, welche neu hinzugekommen sind.

Unterprogramm incxpos

```
; unterprogramm incxkoord  
; "increase xkoord"  
; erhoeht die x-koordinate des sprites  
; um 1, die aktuelle x-koordinate ist  
; in der variablen xpos gespeichert  
  
incxkoord  
    clc  
    lda xkoord  
    adc #$01  
    sta xkoord  
    bcc incxkoordende  
    clc  
    lda xkoord+1  
    adc #$01  
    sta xkoord+1  
  
incxkoordende  
    rts
```

Am Ende des Codes ist folgende Variable definiert:

```
; x-koordinate des sprites  
; wird bei der horizontalen  
; bewegung laufend erhoeht  
  
xkoord    .byte $18,$00
```

Hier wird die aktuelle x-Koordinate des Ballons gespeichert, welche sich während der Bewegung laufend um 1 erhöht. Da es sich bei der x-Koordinate um einen 16 Bit Wert handelt, müssen wir hier zwei Bytes definieren (das erste für das niederwertige und das zweite für das höherwertige Byte)

Das Unterprogramm incxkoord ist dafür zuständig, den Inhalt der Variablen xkoord um 1 zu erhöhen.

Wie eine 16 Bit Addition abläuft, habe ich bereits im Kapitel über Zahlensysteme beschrieben, doch wir werden hier trotzdem die einzelnen Schritte im Detail durchgehen.

Zunächst wird durch den Befehl CLC das Carry Flag zurückgesetzt, da zu Beginn der Addition ja noch kein Übertrag stattgefunden hat.

Als nächstes wird durch den Befehl LDA xkoord das niederwertige Byte der Variablen xkoord in den Akkumulator geladen. Der Befehl ADC #\$01 erhöht den Inhalt des Akkumulators um 1. Das Ergebnis wird sogleich mit dem Befehl STA xkoord in das niederwertige Byte der Variablen xkoord zurückgeschrieben.

Bei der Erhöhung des Akkumulatorinhalts um 1 müssen wir nun zwei Fälle in Bezug auf das Ergebnis unterscheiden.

- 1.) Der Akkumulatorinhalt ist kleiner als 255, d.h. es tritt bei der Erhöhung um 1 kein Übertrag auf (das Carryflag wird also nicht gesetzt)
- 2.) Der Akkumulatorinhalt ist gleich 255, d.h. er springt durch die Erhöhung um 1 auf 0 zurück und das Carryflag wird gesetzt, um den Übertrag anzuzeigen.

Im ersten Fall müssen wir nichts weiter tun, das Programm verzweigt zum Label incxkoordende und kehrt zum Aufrufer zurück.

Im zweiten Fall müssen wir uns durch den Übertrag um das höherwertige Byte der Variablen xkoord kümmern. Zunächst setzen wir das Carryflag wieder zurück, denn ansonsten würde dieses bei der nun folgenden Addition berücksichtigt werden.

Dann laden wir mit dem Befehl LDA xkoord+1 das höherwertige Byte der Variablen xkoord in den Akkumulator und erhöhen dessen Inhalt durch den Befehl ADC #\$01 um 1. Anschließend schreiben wir das Ergebnis in das höherwertige Byte der Variablen xkoord und kehren zum Aufrufer zurück.

Unterprogramm rbreached

```
; unterprogramm rbreached
; "right border reached"
; prueft ob das sprite den rechten
; bildschirmrand (x = 320 bzw. $0140)
; erreicht hat
;
; rueckgabewert im 0 register
; 0 = nicht erreicht, 1 = erreicht

rbreached
    ldy #$00
    lda xkoord
    cmp #$40
    bne rbrende
    lda xkoord+1
    cmp #$01
    bne rbrende
    ldy #$01
rbrende
    rts
```

Dieses Unterprogramm dient zur Überprüfung, ob das Sprite bereits den rechten Bildschirmrand, also die x-Koordinate 320 (\$0140) erreicht hat. Wenn das der Fall ist, legt das Unterprogramm als Ergebnis im Y Register den Wert 1 ab, andernfalls den Wert 0.

Zu Beginn laden wir das Y Register mit dem Wert 0, wir nehmen also mal an, dass das Sprite den rechten Bildschirmrand noch nicht erreicht hat.

Mit dem Befehl LDA xkoord wird das niederwertige Byte der Variablen xkoord in den Akkumulator geladen und durch den nächsten Befehl CMP #\$40 wird geprüft, ob der Inhalt des Akkumulators dem Wert \$40, also dem niederwertigen Byte von 320 (\$0140), entspricht.

Ist dies nicht der Fall, kehrt das Unterprogramm gleich zum Aufrufer zurück und der Wert 0 bleibt unverändert im Y Register.

Entspricht der Inhalt des Akkumulators jedoch dem Wert \$40, dann wird als nächster Schritt das höherwertige Byte der Variablen xkoord in den Akkumulator geladen und mit dem Wert \$01, also dem höherwertigen Byte von 320 (\$0140), verglichen.

Falls hier ebenfalls Gleichheit besteht, dann enthält die Variable xkoord den Wert 320 (\$0140) und das bedeutet, dass das Sprite den rechten Bildschirmrand erreicht hat. Das Y Register wird daher mit dem Wert 1 geladen und das Unterprogramm kehrt zum Aufrufer zurück.

Falls die Werte jedoch unterschiedlich sind, kehrt das Unterprogramm ebenfalls gleich zum Aufrufer zurück und der Wert 0 bleibt unverändert im Y Register.

Unterprogramm delay

```
; unterprogramm delay
;
delay      ldy #$00
loopwait   iny
           cpy #$ff
           bne loopwait
           rts
```

Dieses Unterprogramm dient nur dazu, etwas Zeit verstreichen zu lassen. Das Unterprogramm wird zwischen den einzelnen Bewegungsschritten aufgerufen, damit diese nicht zu schnell abläuft.

Das Y Register wird von 1 bis 255 hochgezählt (was natürlich wie gewollt Zeit verbraucht) und dann kehrt das Unterprogramm zum Aufrufer zurück.

Kommen wir nun zum Assemblercode, der die soeben beschriebenen Unterprogramme aufruft und die horizontale Bewegung des Sprites durchführt.

```
; x-koordinate setzen
; ausgangsposition
; x=24 ($18)

lda #$18
sta $fa ; lo(x) = $18
sta xkoord
lda #$00
sta $fb ; hi(x) = $00
sta xkoord+1

lda #$00
jsr setspritex

; y-koordinate setzen
; bleibt konstant
; y=50 ($32)

lda #$32
sta $fa
lda #$00
jsr setspritey
```

Dieser Abschnitt versetzt das Sprite an die Ausgangsposition in der linken oberen Ecke des Bildschirms. Dies ist die Position $x = 24$ (\$18), $y = 50$ (\$32)

Die x-Koordinate ist wie wir wissen ja ein 16 Bit Wert. In unserem Fall entspricht die x-Koordinate dem Wert 24 (\$0018)

Wir schreiben also als Vorbereitung für den Aufruf des Unterprogramms setspritex das niederwertige Byte von \$0018, also \$18 in die Speicherstelle \$FA und das höherwertige Byte \$00 in die Speicherstelle \$FB.

Die Variable xkoord muss ebenfalls mit diesem Wert initialisiert werden, daher die Befehle STA xkoord und STA xkoord+1, welche den Wert \$18 in das niederwertige Byte und den Wert \$00 in das höherwertige Byte der Variablen xkoord schreiben.

Wir meinen Sprite 0, d.h. wir müssen vor dem Aufruf von setspritex noch den Wert 0 in den Akkumulator schreiben.

Nun müssen wir noch die y-Koordinate setzen, diese beträgt in unserem Fall 50 (\$32). Die y-Koordinate reicht nur bis maximal 255, ist also ein 8 Bit Wert. Wir schreiben die y-Koordinate also in die Speicherstelle \$FA, laden den Akkumulator noch mit der Spritenummer 0 und rufen das Unterprogramm setspritey auf.

```
        ; vor start auf taste  
        ; warten  
loopwaitkey  
        jsr $ffe4  
        beq loopwaitkey
```

Dieser Abschnitt dient dazu, auf einen Tastendruck zu warten. Dazu wird die Kernal Funktion in einer Schleife so lange aufgerufen, bis eine Taste gedrückt wird, im Akkumulator also nach dem Aufruf der Funktion nicht mehr der Wert 0 steht.

Kommen wir nun zum wichtigsten Teil, jenem Teil, in dem das Sprite bewegt wird.

```
        ; sprite horizontal bewegen  
loophoriz  
        ; x-koordinate um 1 erhoeuen  
        jsr incxkoord  
  
        ; sprite auf die neue  
        ; x-koordinate setzen  
  
        lda xkoord  
        sta $fa      ; lo(x) in $fa  
        lda xkoord+1  
        sta $fb      ; hi(x) in $fb  
        lda #$00      ; spritenr im akku  
        jsr setspritex  
  
        ; kurz warten bis zum  
        ; naechsten bewegungsschritt  
  
        jsr delay  
  
        ; hat das sprite  
        ; den rechten bildschirmrand  
        ; erreicht?  
  
        jsr rbreached  
        cpy #$01  
  
        ; wenn nicht dann naechster  
        ; bewegungsschritt nach rechts  
  
        bne loophoriz  
  
        rts
```

In diesem Abschnitt findet nun in einer Schleife die horizontale Bewegung des Sprites statt. Zuerst wird der Inhalt der Variablen xkoord durch Aufruf des Unterprogramms incxkoord um 1 erhöht.

Das niederwertige Byte der neuen Position wird in die Speicherstelle \$FA und das höherwertige Byte des neuen Inhalts in die Speicherstelle \$FB geschrieben. Dann wird noch die Spritenummer 0 in den Akkumulator geladen und durch den Aufruf des Unterprogramms setspritex das Sprite auf seine neue x-Koordinate bewegt.

Durch Aufruf des Unterprogramms warteschleife wird eine kurze Pause eingelegt bis zum nächsten Bewegungsschritt eingelegt.

Als nächstes wird mittels des Unterprogramms rbreached geprüft, ob das Sprite bereits am rechten Bildschirmrand angekommen ist. Steht nach dem Aufruf der Funktion der Wert 0 im Y Register, dann hat das Sprite den rechten Bildschirmrand noch nicht erreicht und die Schleife wird durch Sprung zum Label loophoriz erneut durchlaufen.

Andernfalls ist das Sprite am rechten Bildschirmrand angekommen und das Programm wird beendet.

Soweit so gut, nun sind sie bestens auf die Arbeit mit mehrfarbigen Sprites vorbereitet, weil wir bereits 95% der Arbeit erledigt haben.

Der hauptsächliche Unterschied besteht eigentlich nur darin, dass das Bitmuster, welches wir in unserem Raster aufzeichnen, anders verarbeitet wird als bei den einfarbigen Sprites.

Mehrfarbige Sprites

Bei den mehrfarbigen Sprites reduziert sich die horizontale Auflösung auf die Hälfte, also auf 12 Pixel, da jeweils zwei Bits für die Farbeinstellung eines Pixels gebraucht werden. Hierbei wird in den beiden Bits jedoch nicht direkt eine Farbe angegeben (dafür wären 4 Bits nötig wenn man alle 16 Farben abbilden will), sondern die beiden Bits bilden eine Bitkombination, welche folgende mögliche Werte darstellen kann:

Bitkombination	Farbinformation
00	Hintergrund (transparent)
10	Spritefarbe (Register 53287 - 53294)
01	Mehrfarbenregister #0 (Register 53285)
11	Mehrfarbenregister #1 (Register 53286)

Hier ein Beispiellaster für ein mehrfarbiges Sprite:

	128/64	32/16	8/4	2/1	128/64	32/16	8/4	2/1	128/64	32/16	8/4	2/1
0	00	00	00	00	10	10	10	10	00	00	00	00
1	00	00	00	10	10	10	10	10	10	00	00	00
2	00	00	10	10	10	10	10	10	10	10	00	00
3	00	00	10	10	10	10	10	10	10	10	00	00
4	00	10	10	10	10	10	10	10	10	10	10	00
5	00	10	10	11	10	10	10	10	11	10	10	00
6	00	10	11	11	11	10	10	11	11	11	10	00
7	10	10	11	11	11	10	10	11	11	11	10	10
8	10	10	11	01	11	10	10	11	01	11	10	10
9	10	10	11	01	11	10	10	11	01	11	10	10
10	10	10	10	11	10	10	10	10	11	10	10	10
11	10	10	10	10	10	10	10	10	10	10	10	10
12	10	10	10	10	10	10	10	10	10	10	10	10
13	10	10	10	10	10	10	10	10	10	10	10	10
14	10	10	10	10	10	10	10	10	10	10	10	10
15	10	10	10	10	10	10	10	10	10	10	10	10
16	10	10	10	10	10	10	10	10	10	10	10	10
17	10	10	00	10	10	00	10	10	10	00	10	10
18	10	10	00	10	10	00	00	10	10	00	10	10
19	10	00	00	00	10	00	00	10	00	00	00	10
20	10	00	00	00	10	00	00	10	00	00	00	10

Enthält eine Zelle die Bitkombination 00, dann enthält das Sprite an dieser Stelle keinen Pixel, d.h. hier scheint der Hintergrund durch (in dem Beispiel sind dies die schwarzen Zellen).

Enthält eine Zelle die Bitkombination 10, dann enthält der Pixel an dieser Stelle jene Farbe, welche in dem Farbregister, welches dem Sprite zugeordnet ist, hinterlegt ist. Dies sind dieselben Register wie bei den einfärbigen Sprites, d.h. Register 53287 enthält die Farbinformation für Sprite 0,

Register 53288 enthält die Farbinformation für Sprite 1 usw. bis hin zur Register 53294, welches die Farbinformation für Sprite 7 enthält. In dem Beispiel sind das die grünen Zellen.

Enthält eine Zelle die Bitkombination 01, dann enthält das Sprite an dieser Stelle jene Farbe, welche in dem Mehrfarbenregister #0 (53285) hinterlegt ist. In dem Beispiel sind das die blauen Zellen.

Enthält eine Zelle die Bitkombination 11, dann enthält das Sprite an dieser Stelle jene Farbe, welche in dem Mehrfarbenregister #1 (53286) hinterlegt ist. In dem Beispiel sind das die weißen Zellen.

Die Farben, die in diesen beiden Registern hinterlegt sind, gelten für alle Sprites.

Das heißt, wenn zwei Sprites an derselben Position die Bitkombination 01 oder 11 enthalten, dann haben sie an dieser Stelle auch dieselbe Farbe. Je nach Bitkombination entweder jene aus dem Register 53285 (01) oder jene aus dem Register 53286 (11)

Die Pixel haben im Vergleich zu einfarbigen Sprites jedoch die doppelte Breite, d.h. die sichtbare Breite des Sprites ändert sich trotzdem nicht (24 einfach breite Pixel sind gleich breit wie 12 doppelt breite Pixel)

In horizontaler Richtung haben wir nun nur mehr 12 Zellen, in der vertikalen Richtung hat sich nichts verändert, d.h. es sind hier nach wie vor 21 Zeilen.

Jede Zelle repräsentiert nun jedoch 2 Bits, die jeweils eine der oben genannten Kombinationen annehmen können.

Bei den einfarbigen Sprites entsprach jede Zelle einem Bit (Pixel oder kein Pixel)

Das heißt also, dass wir insgesamt trotzdem wieder auf 24 Bits, also 3 Bytes kommen, weil ja jede der 12 Zellen 2 Bits beinhaltet.

Eine Zeile ist also nach wie vor durch drei Bytes definiert, nur der Inhalt der Bytes wird bei mehrfarbigen Sprites anders interpretiert.

Für die obige Figur lauten die Byte-Werte für die Zeilen:

Erstes Byte	Zweites Byte	Drittes Byte
0	170	0
2	170	128
10	170	160
10	170	160
42	170	168
43	170	232
47	235	248
175	235	250
173	235	122
173	235	122
171	170	234
170	170	170
170	170	170
170	170	170
170	170	170
170	170	170
170	170	170
162	138	138
162	130	138
128	130	2
128	130	2

Gut, dann erstellen wir doch gleich mal ein Programm mit diesem mehrfarbigen Sprite.
Zuerst erfassen wir die Daten aus der Tabelle in DATA-Zeilen:

Hinweis:

Sie können auch vorab das Program spr3bas laden, wenn Sie das Programm nicht selbst abtippen möchten.

```

1000 REM DATEN FUER SPRITE
1010 DATA 0,170,0
1020 DATA 2,170,128
1030 DATA 10,170,160
1040 DATA 10,170,160
1050 DATA 42,170,168
1060 DATA 43,170,232
1070 DATA 47,235,248
1080 DATA 175,235,250
1090 DATA 173,235,122
1100 DATA 173,235,122
1110 DATA 171,170,234
1120 DATA 170,170,170
1130 DATA 170,170,170
1140 DATA 170,170,170
1150 DATA 170,170,170
1160 DATA 170,170,170
1170 DATA 170,170,170
1180 DATA 162,138,138
1190 DATA 162,130,138
1200 DATA 128,130,2
1210 DATA 128,130,2
READY.

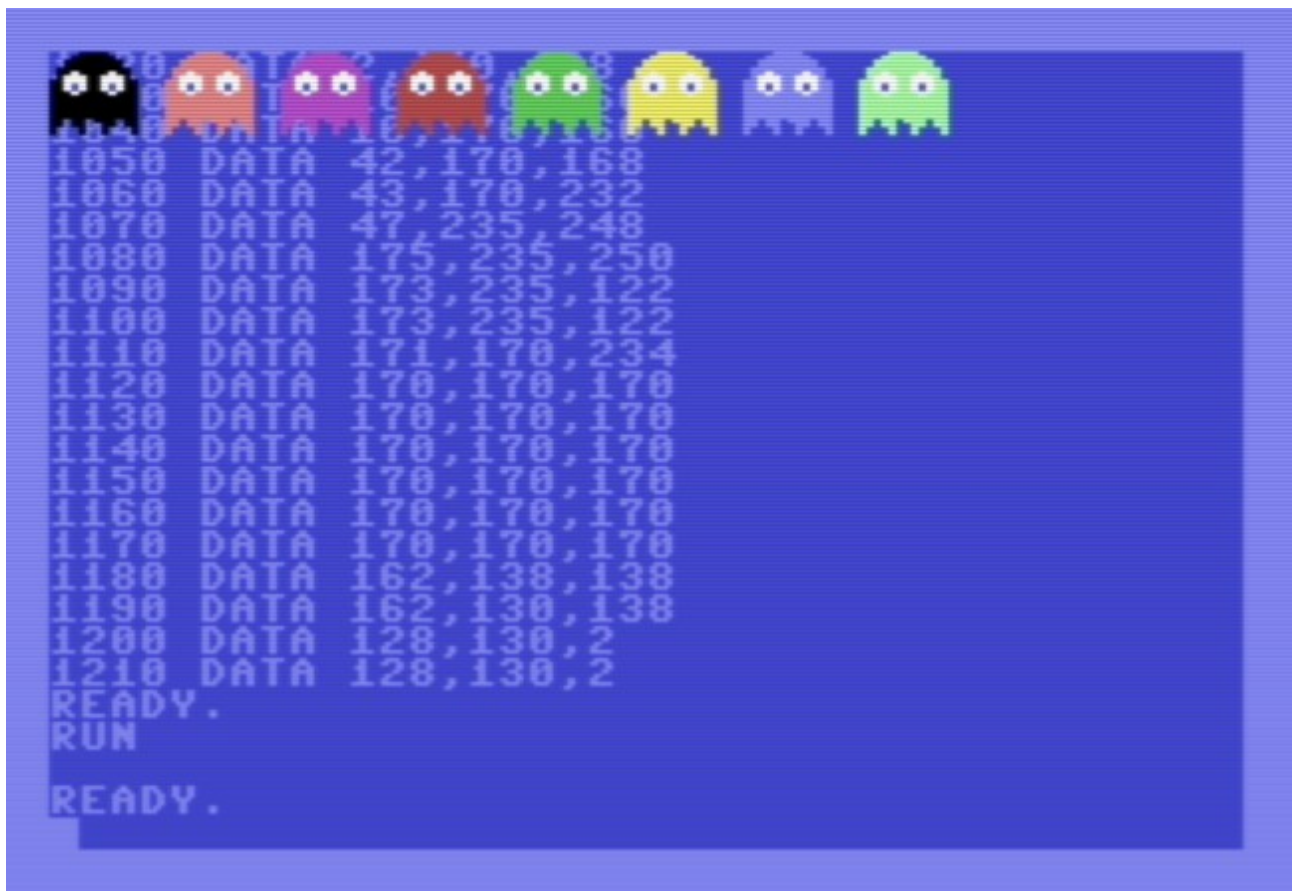
```

Was soll unser Programm machen?

Es soll 8 Sprites, welche sich nur in der Farbgebung unterscheiden, am oberen Rand des Bildschirms anzeigen. Da sich die Sprites nur farblich unterscheiden, ist es nicht nötig, für jedes Sprite eigene Spritedaten zu definieren, sondern wir können für alle Sprites denselben Datenblock nutzen (hier Block Nummer 11) und das individuelle Aussehen der Sprites über die Farbgebung steuern.

Dann sollen sich diese 8 Sprites vom oberen zum unteren Rand des Bildschirms bewegen und danach wieder zurück in die Ausgangsposition am oberen Rand. Um die Bewegung der Sprites zu starten, muss eine beliebige Taste gedrückt werden.

Hier die 8 Geister am Ausgangspunkt bzw. nachdem sie wieder an den oberen Bildschirmrand zurückgekehrt sind:



Gehen wir das BASIC-Programm nun Schritt für Schritt durch.

```
LIST 10-170
10 REM BLOCKNUMMER EINTRAGEN
20 FOR I=2040 TO 2047
30 POKE I,11
40 NEXT I
50 REM SPRITEDATEN IN BLOCK 11 KOPIEREN
60 FOR I=0 TO 62
70 READ A
80 POKE 704+I,A
90 NEXT I
100 REM ALLE SPRITES SIND MEHRFARBIG
110 POKE 53276,255
120 REM ALLE SPRITES VOR HINTERGRUND
130 POKE 53275,0
140 REM ERSTE GEMEINSAME FARBE
150 POKE 53285,6
160 REM ZWEITE GEMEINSAME FARBE
170 POKE 53286,1
READY.
```

Zeile 10 – 40

Hier wird für jedes Sprite dieselbe Blocknummer (11) eingetragen, da ja jedes Sprite gleich aussieht und die Unterschiede nur in der Farbgebung bestehen.

Zeile 50 – 90

Hier werden die Spritedaten aus den DATA-Zeilen in den Speicherblock 11 kopiert.

Zeile 100 – 110

Da alle Sprites mehrfarbig sind, können wir in Speicherstelle 53276 alle 8 Bits auf den Wert 1 setzen, also den Wert 255 dort eintragen.

Zeile 120 – 130

Alle Sprites sollen sich vor dem Hintergrund befinden, also höhere Priorität als der Hintergrund haben. Daher setzen wir alle 8 Bits auf den Wert 0 und tragen den Wert 0 in die Speicherstelle 53275 ein.

Zeile 140 – 150

Hier wird die erste Farbe, welchen allen Sprites gemeinsam ist, eingestellt. In unserem Beispiel hier ist das die Farbe Blau (Farbcode 6). Dies ist die Augenfarbe der Geister.

Zeile 160 – 170

Hier wird die zweite Farbe, welchen allen Sprites gemeinsam ist, eingestellt. In unserem Beispiel hier ist das die Farbe Weiß (Farbcode 1). Dies ist „das Weiße“ im Auge der Geister.

Zeile 180 – 330

Hier wird die individuelle Farben für jedes der 8 Sprites eingestellt.

```
LIST 180-330
180 REM FARBE FUER SPRITE 0
190 POKE 53287,0
200 REM FARBE FUER SPRITE 1
210 POKE 53288,10
220 REM FARBE FUER SPRITE 2
230 POKE 53289,4
240 REM FARBE FUER SPRITE 3
250 POKE 53290,2
260 REM FARBE FUER SPRITE 4
270 POKE 53291,5
280 REM FARBE FUER SPRITE 5
290 POKE 53292,7
300 REM FARBE FUER SPRITE 6
310 POKE 53293,14
320 REM FARBE FUER SPRITE 7
330 POKE 53294,13

READY.
```

Zeile 340 – 490

Hier werden sowohl die X- als auch die Y – Koordinate für jedes der 8 Sprites eingestellt. Zu Beginn befinden sich alle 8 Sprites am oberen Bildschirmrand, daher ist zu Beginn die Y – Koordinate bei allen Sprites gleich.

```
LIST 340-510
```

```
340 REM POSITION VON SPRITE 0
350 POKE 53248,24:POKE 53249,50
360 REM POSITION VON SPRITE 1
370 POKE 53250,55:POKE 53251,50
380 REM POSITION VON SPRITE 2
390 POKE 53252,86:POKE 53253,50
400 REM POSITION VON SPRITE 3
410 POKE 53254,117:POKE 53255,50
420 REM POSITION VON SPRITE 4
430 POKE 53256,148:POKE 53257,50
440 REM POSITION VON SPRITE 5
450 POKE 53258,179:POKE 53259,50
460 REM POSITION VON SPRITE 6
470 POKE 53260,210:POKE 53261,50
480 REM POSITION VON SPRITE 7
490 POKE 53262,241:POKE 53263,50
500 REM ALLE SPRITES AKTIVIEREN
510 POKE 53269,255
```

```
READY.
```

Zeile 500 – 510

Hier werden alle Sprites aktiviert, also auf den Positionen, die wir eingestellt haben, angezeigt. Da wir alle 8 Sprites aktivieren wollen, ist es nicht nötig, jedes einzelne zu aktivieren, sondern wir können alle Sprites in einem Schwung sichtbar machen, in dem wir alle 8 Bits in der Speicherstelle 53269 auf den Wert 1 setzen, also den Wert 255 dorthin schreiben.

Zeile 520 – 790

Hier findet die Bewegung der Sprites nach unten bzw. anschließend wieder nach oben statt.

Die Werte für die Y – Koordinate werden jeweils in einer Schleife durchlaufen und durch das Unterprogramm ab Zeile 700 werden die aktuellen Y - Koordinaten für jedes Sprite aktualisiert.

```
LIST 520-790
520 REM SPRITES NACH UNTEN BEWEGEN
530 FOR Y=51 TO 229
540 GOSUB 700
550 NEXT Y
560 REM SPRITES NACH OBEN BEWEGEN
570 FOR Y=228 TO 50 STEP-1
580 GOSUB 700
590 NEXT Y
600 END
700 REM Y KOORDINATEN SETZEN
710 POKE 53249,Y
720 POKE 53251,Y
730 POKE 53253,Y
740 POKE 53255,Y
750 POKE 53257,Y
760 POKE 53259,Y
770 POKE 53261,Y
780 POKE 53263,Y
790 RETURN
READY.
```

Anmerkung: Das Warten auf einen Tastendruck zu Beginn des Programms habe ich erst nachträglich eingebaut, d.h. es muss noch folgende Zeile 515 ergänzt werden:

```
510 POKE 53269,255
515 GET AS:IF AS="" THEN 515
520 REM SPRITES NACH UNTEN BEWEGEN
```

Starten Sie das Programm mit RUN, drücken eine beliebige Taste und sehen Sie zu, wie sich die 8 mehrfarbigen Sprites gemächlich von oben nach unten bzw. anschließend wieder nach oben bewegen.

Nun wird es wieder spannend, denn wir wollen die Assembler-Version dieses BASIC-Programms in Angriff nehmen.

Starten Sie den TMP und laden die Datei spr3asm in den Editor. Wenn Sie das Programm starten und eine beliebige Taste drücken, fällt als erstes der enorme Geschwindigkeitsunterschied zur BASIC-Version auf.

Und dabei ist bei der Assembler-Version zwischen jedem Bewegungsschritt eine Pause enthalten, damit man die Bewegung der Sprites überhaupt mitverfolgen kann. Bei der BASIC-Version brauchen wir diese Pause zwischen den Bewegungsschritten nicht, da es hier sowieso schon recht gemütlich zugeht.

Die BASIC-Version benötigte bei mir ca. 25 Sekunden an Laufzeit, die Assembler-Version war in nicht mal 2 Sekunden durchgelaufen. Und das trotz der Pausen zwischen den Bewegungsschritten!

Man sieht an diesem Beispiel also wieder einmal sehr eindrucksvoll den enormen Geschwindigkeitsunterschied zwischen BASIC und Assembler.

Beginnen wir mit den Daten am Ende des Programms. Die Namen werden im Assemblercode, welcher vor den Daten im Programm steht, verwendet und damit Sie beim Lesen des Codes bereits wissen, was gemeint ist, möchte ich mit der Erklärung hier beginnen.

Den Anfang bildet hier das Feld mit den Spritedaten (pacmangeist)

Es folgt ein Datenfeld mit den individuellen Farben der Sprites (spritefarben) und schließlich noch zwei einzelne Variablen für die beiden gemeinsamen Farben (gemfarbe1 und gemfarbe2)

```
;-----  
; spritedaten  
pacmangeist  
    .byte $00,$aa,$00  
    .byte $02,$aa,$80  
    .byte $0a,$aa,$a0  
    .byte $0a,$aa,$a0  
    .byte $2a,$aa,$a8  
    .byte $2b,$aa,$e8  
    .byte $2f,$eb,$f8  
    .byte $af,$eb,$fa  
    .byte $ad,$eb,$7a  
    .byte $ad,$eb,$7a  
    .byte $ab,$aa,$ea  
    .byte $aa,$aa,$aa  
    .byte $aa,$aa,$aa  
    .byte $aa,$aa,$aa  
    .byte $aa,$aa,$aa  
    .byte $aa,$aa,$aa  
    .byte $a2,$8a,$8a  
    .byte $a2,$82,$8a  
    .byte $80,$82,$02  
    .byte $80,$82,$02  
  
;-----  
; individuelle farben der sprites  
spritefarben  
    .byte $00,$02,$03,$04  
    .byte $05,$07,$0a,$0d  
  
;-----  
; gemeinsame farbe 1  
gemfarbe1  
    .byte $06  
  
;-----  
; gemeinsame farbe 2  
gemfarbe2  
    .byte $01
```

Kommen wir nun zum Assembler-Code.

Die Sprites unterscheiden sich nur in der Farbe, haben aber ansonsten dasselbe Aussehen, d.h. wir brauchen nur einen Datenblock mit den Spritedaten und tragen in die Speicherstellen, welche die Blocknummer für die einzelnen Sprites enthalten, einfach dieselbe Nummer ein.

Wir verwenden gleich den Block 11.

```

; blocknr. 11 ($0b) fuer alle
; sprites einstellen, d.h.
; $0b in die speicherstellen
; 2040 ($07f8) - 2047 ($07ff)
; schreiben

ldx #$00
lda #$0b
loopblocknr
sta $07f8,x
inx
cpx #$08
bne loopblocknr

```

Anschließend kopieren wir die Spritedaten in diesen Block.

```

; spritedaten kopieren
; da alle sprites gleich
; aussehen und sich nur durch
; die farbe unterscheiden,
; brauchen wir nur einen block
; fuer die spritedaten.
; dies ist der oben fuer alle
; sprites eingestellte
; block 11 ($0b)
; dieser beginnt bei adresse
; 11 * 64 = 704 ($02c0)

copyloop
ldx #$00
lda pacmangeist,x
sta $02c0,x
inx
cpx #$3f
bne copyloop

```

Dann aktivieren wir alle Sprites.

```

; alle sprites einschalten

lda #$ff
sta $d015

```

Da wir nun ja mit mehrfarbigen Sprites arbeiten, definieren wir alle Sprites als mehrfarbig. Dazu müssen wir in der Speicherstelle \$D01C alle Bits auf 1 setzen.

```

; alle sprites sind mehrfarbig

lda #$ff
sta $d01c

```

Als nächstes setzen wir die individuellen Farben der Sprites. Dazu müssen wir in die Speicherstellen \$D027 (Sprite 0) bis \$D02E (Sprite 7) den gewünschten Farbcode eintragen.

```

; individuelle spritefarben
; setzen

ldx #$00
loopsetfarbe
lda spritefarben,x
sta $d027,x
inx
cpx #$08
bne loopsetfarbe

```

Nun müssen wir die beiden Farben festlegen, die alle Sprites gemeinsam haben.

```
    ; gemeinsame farbe 1 setzen
    lda gemfarbe1
    sta $d025

    ; gemeinsame farbe 2 setzen
    lda gemfarbe2
    sta $d026
```

Nun legen wir noch fest, dass die Sprites vor dem Hintergrund liegen, also Priorität gegenüber diesem haben.

```
    ; alle sprites vor hintergrund
    lda #$00
    sta $d01b
```

Der nächste Schritt besteht darin, die Sprites an ihren Ausgangspositionen anzuzeigen.

Zuerst setzen wir die x-Koordinaten durch Schreiben der gewünschten Werte in die entsprechenden Speicherstellen.

```
    ; sprites an die
    ; ausgangspositionen
    ; setzen

    ; x-koordinaten der sprites
    ; setzen

    ldx #$10
    stx $d000 ; x fuer sprite 0

    ldx #$37
    stx $d002 ; x fuer sprite 1

    ldx #$56
    stx $d004 ; x fuer sprite 2

    ldx #$75
    stx $d006 ; x fuer sprite 3

    ldx #$94
    stx $d008 ; x fuer sprite 4

    ldx #$b3
    stx $d00a ; x fuer sprite 5

    ldx #$d2
    stx $d00c ; x fuer sprite 6

    ldx #$f1
    stx $d00e ; x fuer sprite 7
```

Nun müssen wir noch die y-Koordinaten der Sprites angeben. Da in diesem Programm die Sprites alle immer auf derselben y-Koordinate angezeigt werden, habe ich zu diesem Zweck ein Unterprogramm setyforsprites ergänzt, da wir diese Aufgabe mehrmals benötigen.

Vor dem Aufruf des Unterprogramms schreibt man die gewünschte y-Koordinate in das Y Register und innerhalb des Unterprogramms wird dieser Wert dann für alle Sprites eingestellt.

```
;-----  
; unterprogramm setyforsprites  
; setzt die y-koordinate fuer alle  
; sprites  
;  
; parameter:  
; y register: y koordinate  
  
setyforsprites  
    sty $d001 ; y fuer sprite 0  
    sty $d003 ; y fuer sprite 1  
    sty $d005 ; y fuer sprite 2  
    sty $d007 ; y fuer sprite 3  
    sty $d009 ; y fuer sprite 4  
    sty $d00b ; y fuer sprite 5  
    sty $d00d ; y fuer sprite 6  
    sty $d00f ; y fuer sprite 7  
    rts
```

Zu Beginn müssen wir alle Sprites am oberen Bildschirmrand positionieren, d.h. wir rufen das Unterprogramm mit dem Wert 50 (\$32) im Y Register auf.

```
; alle sprites starten  
; am oberen bildschirmrand  
; y = 50 ($32)  
  
ldy #$32  
jsr setyforsprites
```

Nun sind die Sprites schon mal auf ihren Ausgangspositionen zu sehen.

Damit die Bewegung nicht unmittelbar startet, habe ich mittels der bereits bekannten Kernal-Funktion \$FFE4 wiederum das Warten auf eine beliebige Taste eingebaut.

```
; vor start auf taste  
; warten  
  
loopwaitkey  
    jsr $ffe4  
    beq loopwaitkey
```

Nun werden die Sprites schrittweise vom oberen zum unteren Bildschirmrand bewegt.

Wir verwenden das Y Register um während der Bewegung die aktuelle y-Koordinate der Sprites zu speichern.

Zu Beginn schreiben wir den Wert \$32 in das Y Register, da sich die Sprites am oberen Bildschirmrand befinden.

In der Schleife loopdown zählen wir laufend das Y Register hoch und rufen mit dem neuen Inhalt das Unterprogramm setyforsprites auf. Dies bewirkt, dass alle Sprites um eine Position nach unten wandern.

Dann wird durch Aufruf des Unterprogramms delay eine kleine Pause bis zum nächsten Bewegungsschritt eingelegt.

```
;-----  
; unterprogramm delay  
; zaehlt im x register von 0 bis 255  
; hoch, damit etwas zeit vergeht  
  
delay      ldx #$00  
  
loopwait   inx  
           cpx #$ff  
           bne loopwait  
           rts
```

Anschließend wird geprüft, ob die Sprites bereits den unteren Bildschirmrand erreicht haben. Dies ist dann der Fall, wenn im Y Register der Wert \$e5 steht.

```
; sprites zum unteren  
; bildschirmrand bewegen  
;  
; das y register enthaelt  
; waehrend der bewegung  
; immer die aktuelle  
; y-koordinate der sprites  
; wir starten am oberen  
; bildschirmrand: y = 50 ($32)  
; und bewegen uns bis zum  
; unteren bildschirmrand:  
; y = 229 ($e5)  
  
ldy #$32  
  
loopdown   ; y-koordinate um 1 erhoehen  
           iny  
           ; y-koordinaten der sprites  
           ; aktualisieren  
           jsr setyforsprites  
           ; kurz warten bis zum  
           ; naechsten bewegungsschritt  
           jsr delay  
           ; haben die sprites  
           ; den unteren bildschirmrand  
           ; erreicht?  
           cpy #$e5  
           ; wenn nicht dann naechster
```



```
    ; bewegungsschritt nach unten  
    bne loopdown
```

Nun sollen die Sprites wieder zurück zum oberen Bildschirmrand bewegt werden. Dies erfolgt auf dieselbe Art und Weise wie die Bewegung zum unteren Bildschirmrand, nur dass hier die y-Koordinate nicht hochgezählt, sondern bis zum Wert \$32 (oberer Bildschirmrand erreicht) runtergezählt wird.

```
    ; sprites wieder zum oberen  
    ; bildschirmrand bewegen  
loopup    ; y-koordinate um 1 vermindern  
          dey  
          ; y-koordinaten der sprites  
          ; aktualisieren  
          jsr setyforsprites  
          ; kurz warten bis zum  
          ; naechsten bewegungsschritt  
          jsr delay  
          ; haben die sprites den  
          ; oberen bildschirmrand  
          ; erreicht?  
          cpy #$32  
          ; wenn nicht dann naechster  
          ; bewegungsschritt nach oben  
          bne loopup  
          rts
```

Sobald die Sprites wieder am oberen Bildschirmrand angekommen sind, wird die Schleife loopup verlassen und das Programm durch den Befehl RTS beendet.