

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?

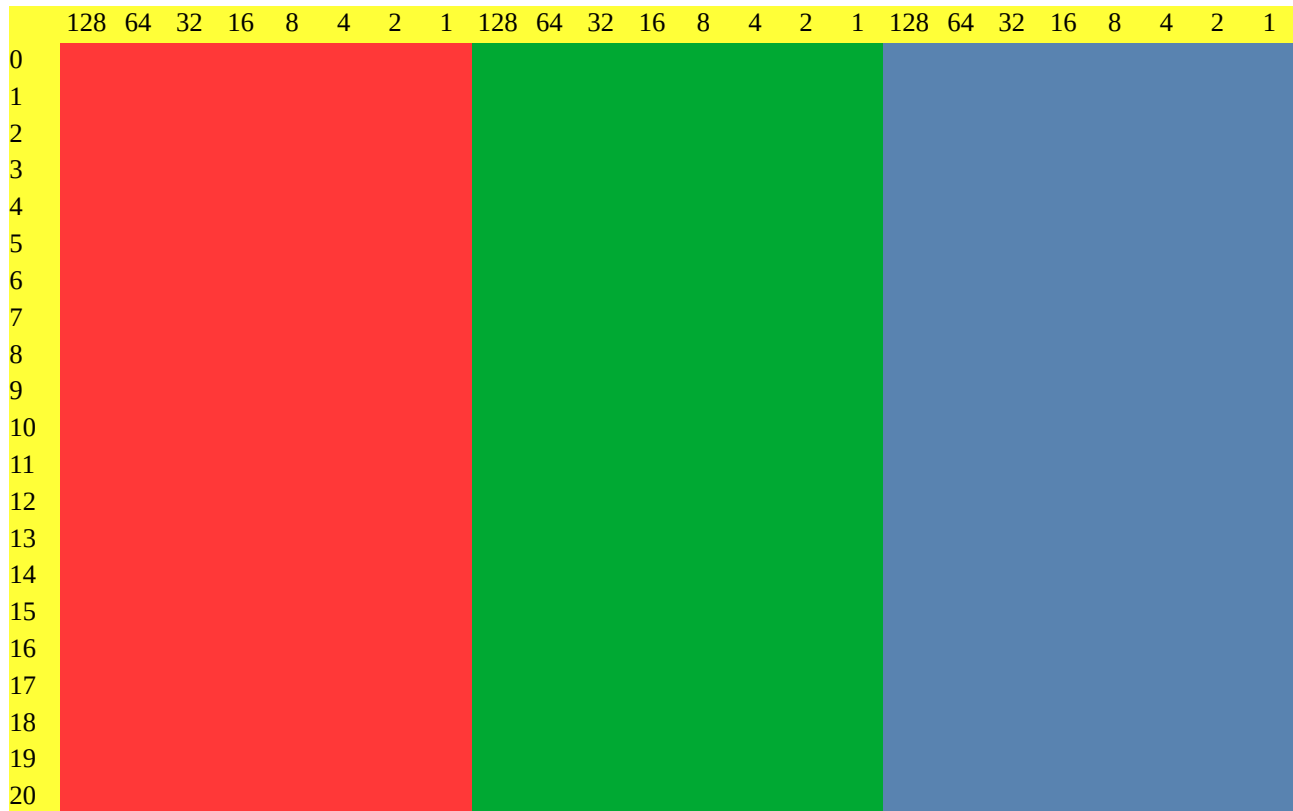
Dazu zeichnet man sich zunächst beispielsweise auf kariertem Papier einen Raster mit 24 Spalten und 21 Zeilen auf, wobei jede Zelle des Rasters einem der 504 Pixel des Sprites entspricht.

Das Sprite hat eine horizontale Auflösung von 24 Pixel und wenn man jedem Pixel ein Bit zuordnet, dann benötigen wir 3 Bytes (3×8 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 (3×21 Bytes = 63 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.

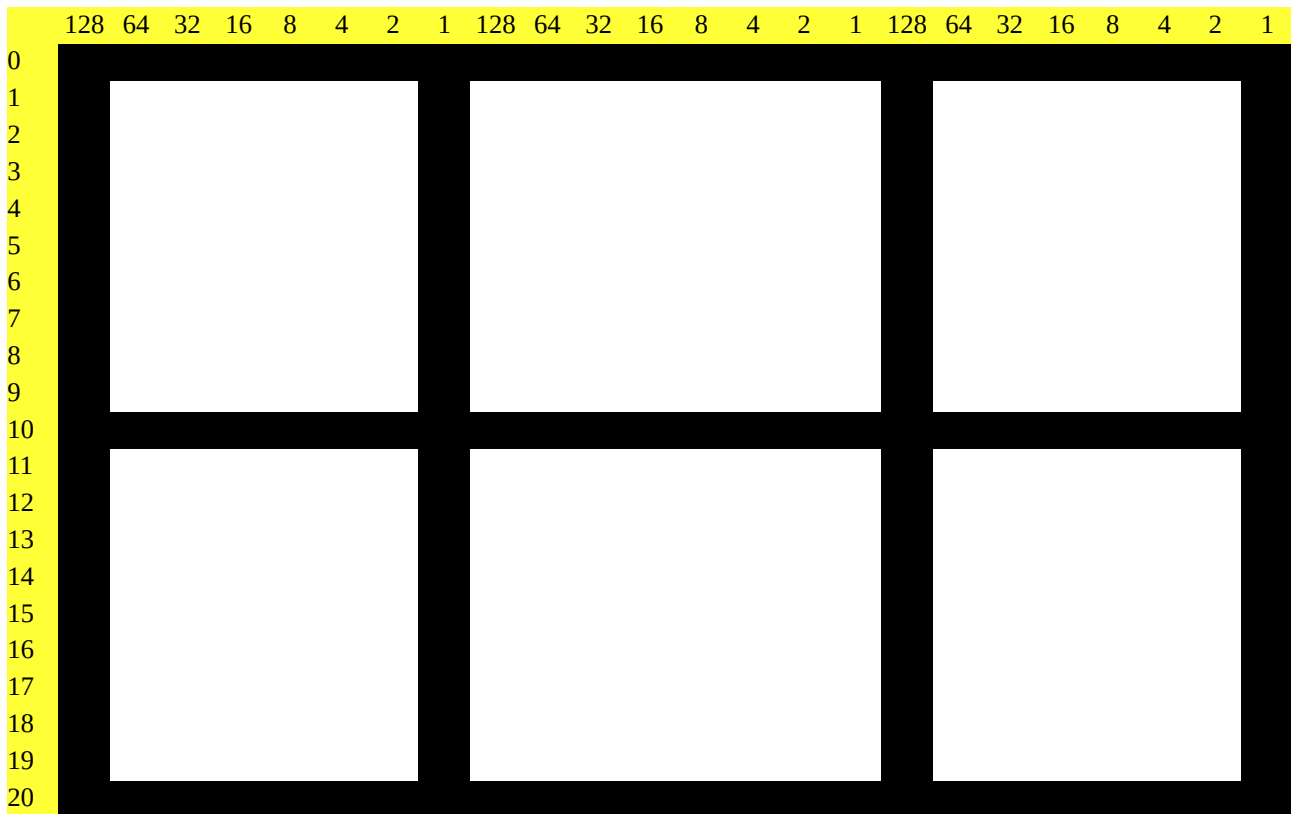
Über jedes Bit der drei Bytes schreiben wir die jeweilige Wertigkeit an der Stelle.

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.

```

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.

Spiele 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.

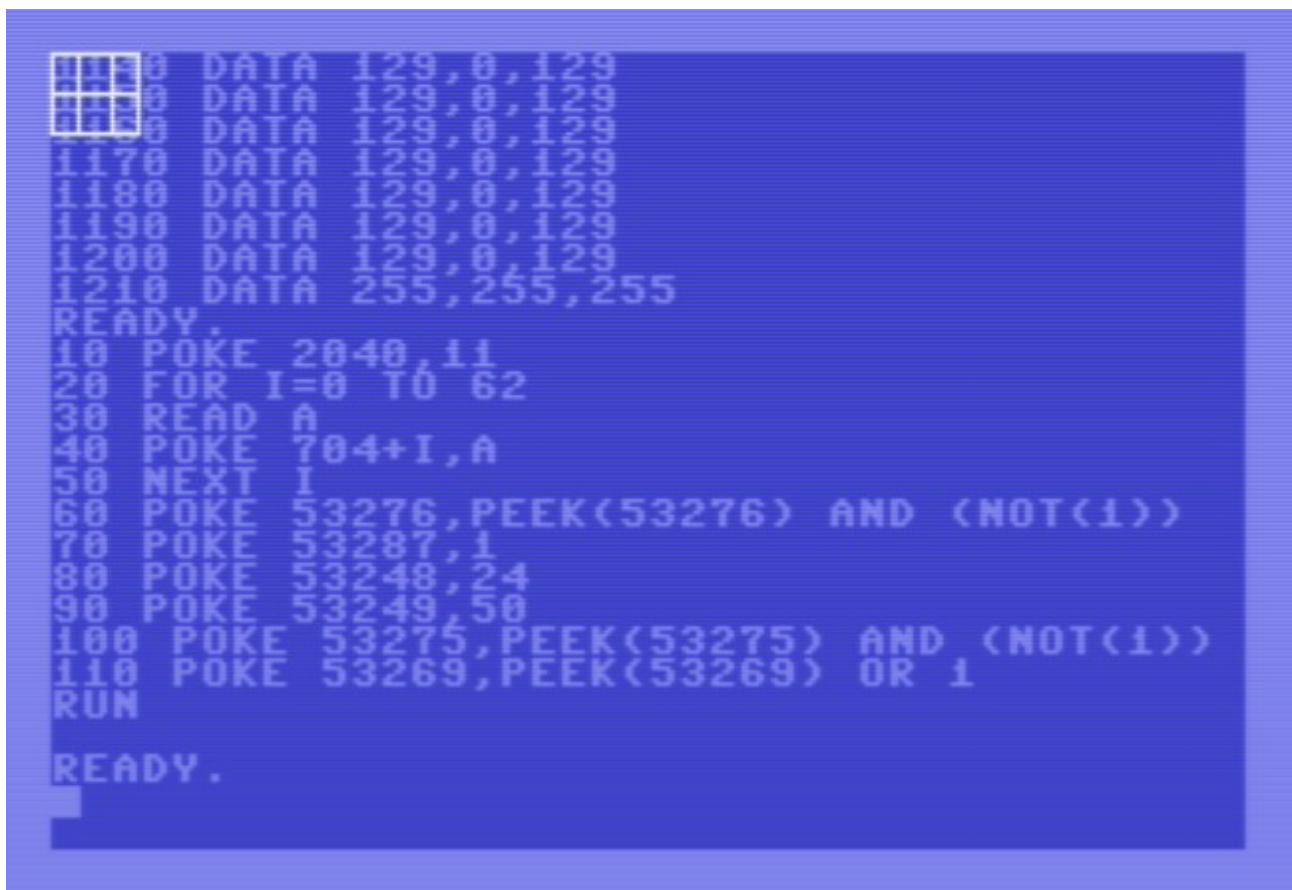
Wichtig:

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

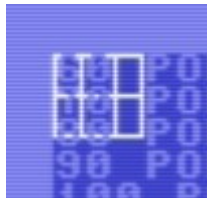
```
POKE 53269,PEEK(53269) OR 1
```

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.

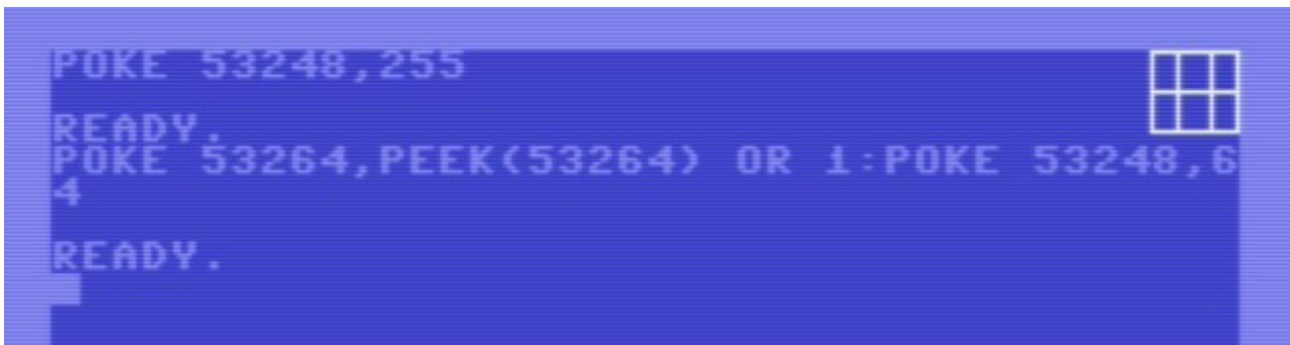


Wollen wir das Sprite an den rechten Rand des sichtbaren Bereichs positionieren, also auf die Position 320, dann müssen wir die Speicherstelle 53264 zu Hilfe nehmen.

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.

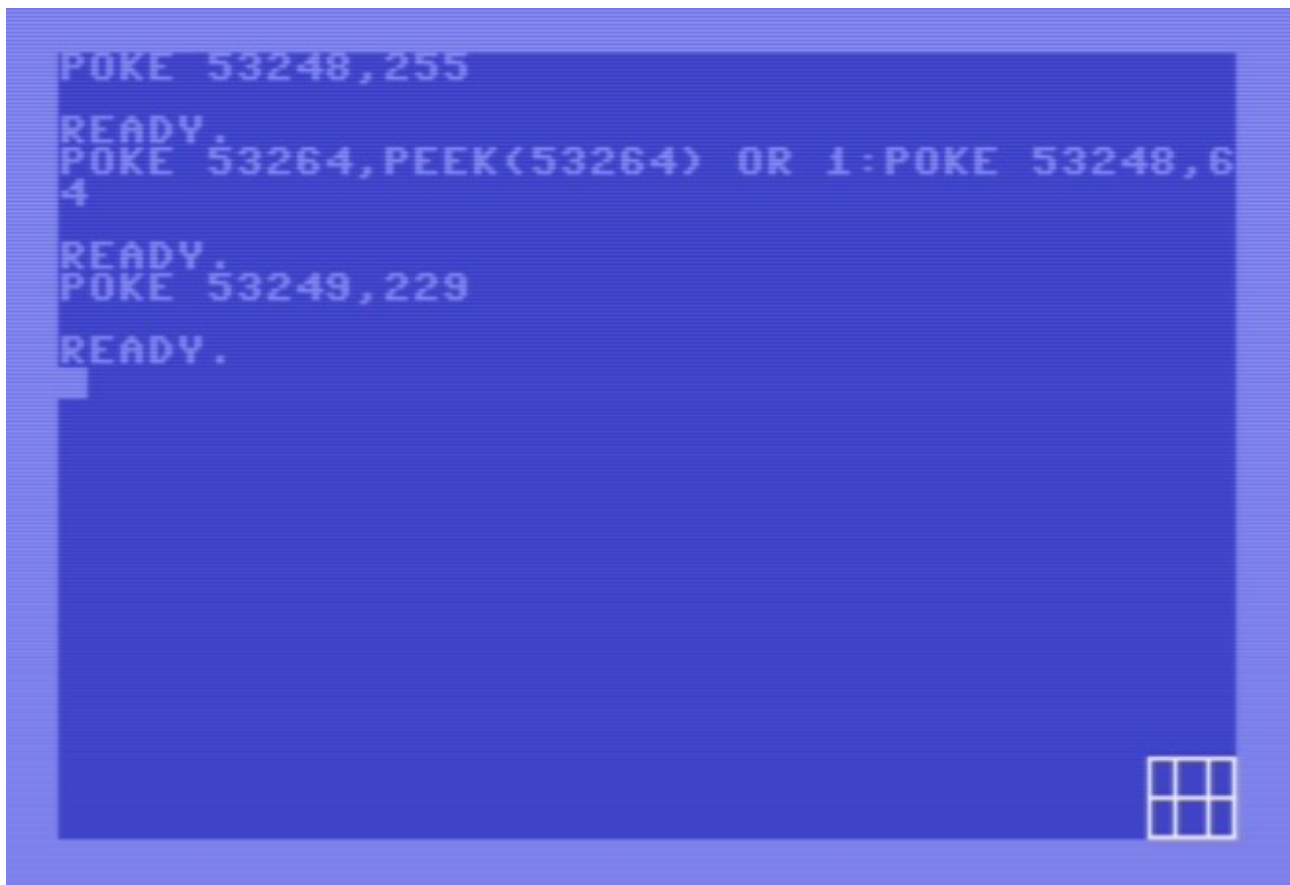
```
POKE 53264,PEEK(53264) OR 1:POKE 53248,64
```



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:



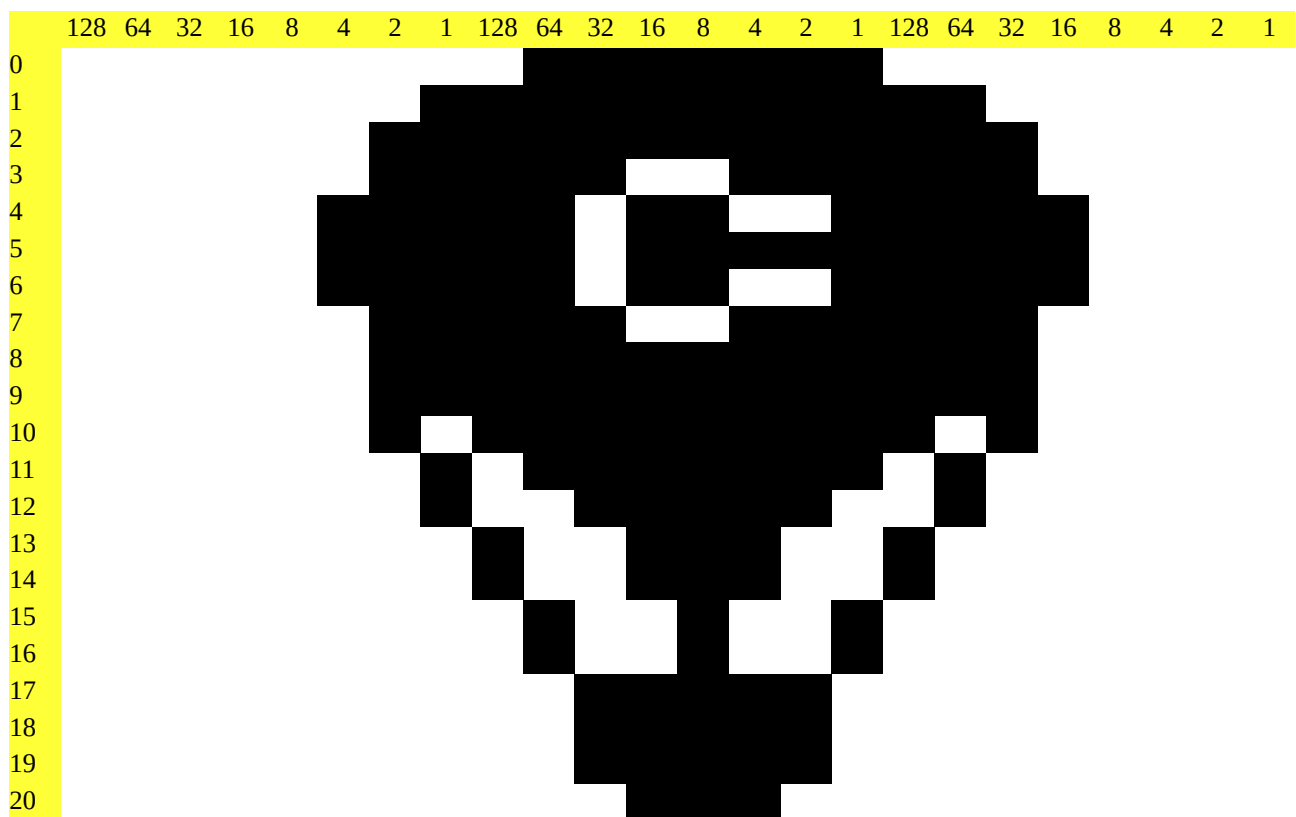
Für eine vertikale Vergrößerung ist die Speicherstelle 53271 zuständig. Auch hier ist wieder jedes Sprite mit einem eigenen Bit vertreten. Setzt man es auf den Wert 1, so wird das Sprite in vertikaler Richtung verdoppelt. Setzt man es umgekehrt auf den Wert 0, so wird das Sprite in Normalgröße angezeigt.

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

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

```

1210 DATA 255,255,255
READY.
1220 REM DATEN FUER 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.



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.

Laden Sie dazu den Sourcecode sprite1src in den Editor, sodass wir ihn gemeinsam Schritt für Schritt durchgehen können.

Da ich versucht habe, das Programm gut zu dokumentieren, sollte der Sourcecode fast selbsterklärend sein.

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)

```
        ; 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.

```
copyloop    ldx #$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 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.

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

```
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 Bereich 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)

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

```
setspritex
    pha
    asl a
    tay
    lda $fa
    sta $d000,y
    lda $fb
    beq xk1g255
    pla
    sec
    jsr setbit8forx
    jmp setxende
xk1g255
    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 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 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

```
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.

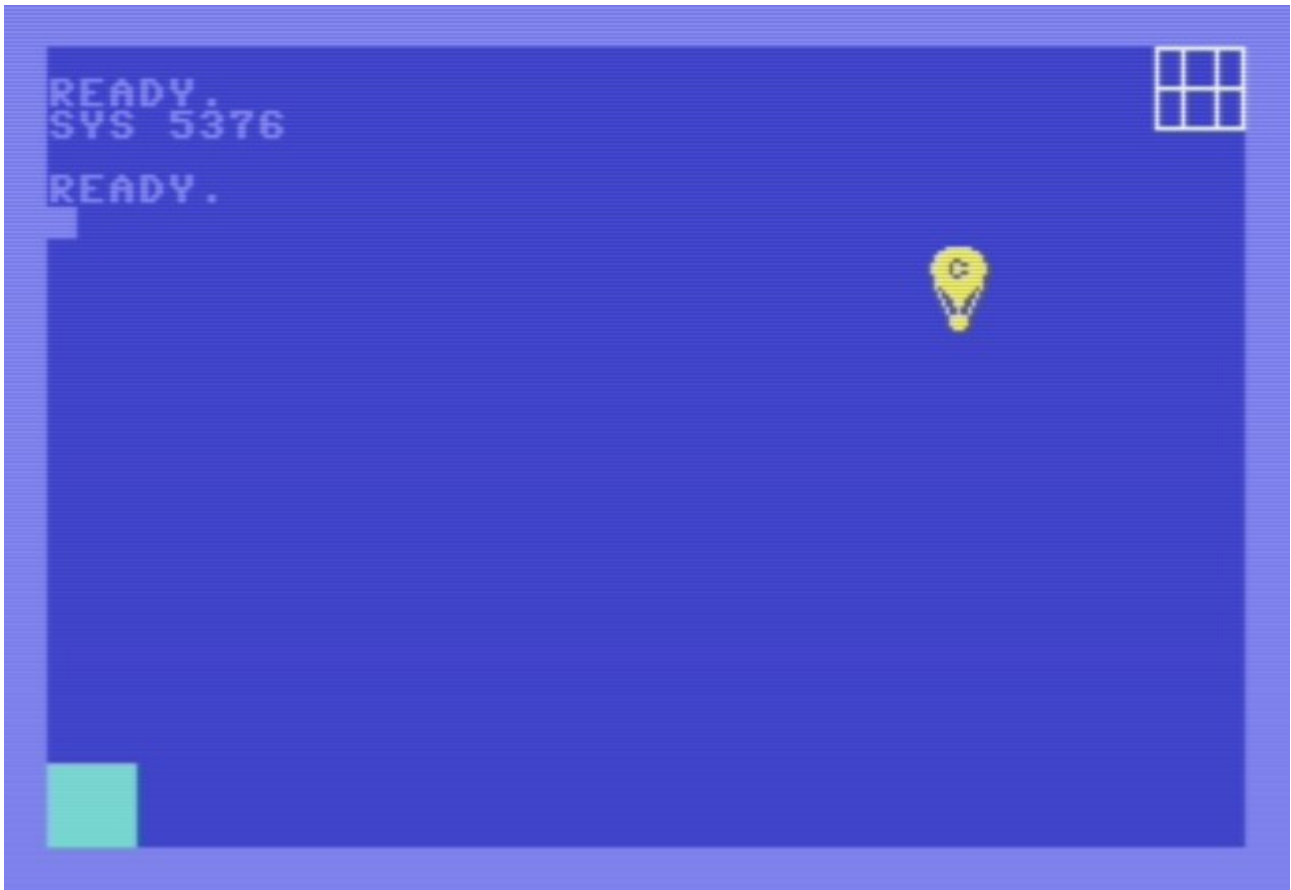
Dann schreiben wir die y-Koordinate in die Speicherstelle \$fa, laden den Akkumulator mit der Spritenummer 0 und rufen das Unterprogramm setspritey auf.

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 wieder vieles von dem an, was wir bisher gelernt haben.

Wir werden ein Programm schreiben, das ein Sprite vom linken zum rechten Rand des Bildschirms bewegt.

Dadurch werden wir den Mechanismus zur Bildung der x-Koordinate von Sprites nochmals verinnerlichen und im Zuge dessen werden wir auch der 16 Bit Addition begegnen.