

Ein Sprite-Editor als Abschlussprojekt

Da ich dieses Buch nicht mit bloßer Theorie abschließen möchte, würde ich Ihnen gerne die Programmierung eines größeren Programms in Assembler demonstrieren. Ich habe mich, passend zum vorherigen Kapitel, für einen Sprite-Editor entschieden.

Damit Sie sich ein Bild vom Endergebnis unserer Bemühungen machen können, sehen Sie hier ein Bild des Sprite-Editors.



Am besten ist es jedoch, wenn Sie das Programm mit LOAD „SPRITEEDITOR“,8,1 von der Diskette laden und mit SYS 12288 starten. Auf diese Art und Weise lernen Sie das Programm kennen und können es natürlich auch zur Erstellung von Sprites nutzen. In der jetzigen Form ist der Editor auf einfarbige Sprites ausgelegt, aber mit den Kenntnissen, welche Sie nach Abschluss des Projekts erlangt haben, ist es für Sie sicher ein Leichtes, eine entsprechende Erweiterung umzusetzen :)

Hier eine kurze Bedienungsanleitung:

- **Cursortasten:** Sie ermöglichen die Bewegung des Cursors innerhalb des Editorbereichs.
- **RETURN-Taste:** Setzt ein Bit an der aktuellen Cursorposition und bewegt den Cursor um eine Position nach rechts.

- **Leertaste:** Löscht das Bit an der aktuellen Cursorposition und bewegt den Cursor um eine Position nach rechts.
- **SHIFT + S:** Das Sprite wird in eine Datei namens SPRITE auf der Diskette gespeichert.

Aktuell ist es noch nicht möglich, beim Speichern einen Dateinamen zu vergeben, sondern es wird der vorgegebene Name SPRITE für die Datei verwendet. Dies war jedoch Absicht von mir, damit noch Raum für Verbesserungen bleibt. Um das Sprite also nicht durch das nächste Sprite zu überschreiben, müssen Sie die Datei SPRITE vor der Erstellung eines neuen Sprites in einen Namen Ihrer Wahl umbenennen.

ACHTUNG: Falls die Datei SPRITE bereits auf der Diskette existiert, wird sie ohne Sicherheits-Rückfrage überschrieben. Auch dieser Punkt fällt in den absichtlich von mir gelassenen Raum für Verbesserungen.

Dasselbe gilt für die folgenden beiden Funktionen, auch hier erfolgt keine Sicherheits-Rückfrage, sodass das aktuell angezeigte Bitmuster überschrieben bzw. gelöscht wird.

- **SHIFT + L:** Das Sprite wird aus der Datei SPRITE wieder von der Diskette geladen und im Editor angezeigt.
- **SHIFT + CLR/HOME:** Der Editorbereich wird gelöscht.
- **SHIFT + H:** Im oberen Teil des Bildschirms wird ein Fenster mit Hilfefunktionen eingeblendet. Dieses Fenster verschwindet, sobald Sie eine beliebige Taste drücken.
- **SHIFT + Q:** Beendet den Sprite-Editor und kehrt zu BASIC zurück. Natürlich können Sie ihn jederzeit wieder mit SYS 12288 aufrufen.

Vorab eine Anmerkung in Bezug auf den Begriff „Cursor“

Einerseits gibt es den allbekannten, blinkenden C64 Cursor und andererseits verfügt der Sprite-Editor über einen eigenen Cursor, der nichts mit dem C64 Cursor zu tun hat. An jenen Stellen, an denen ich den C64 Cursor meine, werde ich daher explizit darauf hinweisen.

Auf der Diskette gibt es noch ein BASIC-Programm namens SHOWSPRITE. Dieses demonstriert, wie die Spritedaten, welche in der Datei gespeichert sind, dann konkret als Sprite auf den Bildschirm gebracht werden können.

Es steckt keine Hexerei dahinter, die Datei wird einfach Byte für Byte eingelesen und die Bytes in einem Array gespeichert. Nachdem alle Bytes eingelesen wurden, werden über die bereits bekannten POKE-Befehle die Einstellungen für das Sprite getroffen und dieses auf dem Bildschirm angezeigt.

Nachdem Sie nun mit dem Sprite-Editor vertraut sind, können wir uns der Programmierung widmen. Ich habe die Umsetzung auf zwei große Teile aufgeteilt. Der erste Teil umfasst die Darstellung des Editors am Bildschirm, die Cursorfunktionen sowie das Setzen und Löschen von Bits.

Im zweiten Teil werden wir dann die restlichen Programmfunktionen wie beispielsweise das Speichern und Laden des Sprites umsetzen.

Beginnen wir also mit dem ersten Teil.

Zunächst müssen wir uns einige elementare Grundfunktionen in Form von Unterprogrammen zusammenstellen, die wir im Laufe der Entwicklung des Sprite-Editors immer wieder benötigen werden.

Für die ersten Schritte erstellen wir folgende Unterprogramme:

- **asl16:** Stellt eine 16bit Version des Befehls ASL dar, es bietet also die Möglichkeit eine 16bit Zahl bitweise um eine bestimmte Anzahl an Stellen nach links verschieben zu können.
- **adc16:** Stellt eine 16bit Version des Befehls ADC dar, dadurch ist es also möglich, zwei 16bit Zahlen zu addieren.
- **calcposaddr:** Berechnet aus einer Position am Bildschirm (gegeben durch einen Zeilen- und einen Spaltenwert) die entsprechende Adresse im Bildschirmspeicher. Hier kommen die vorherigen beiden Unterprogramme oft zur Anwendung.

Anmerkung: Die Unterprogramme, die Sie hier sehen werden, stammen 1:1 aus dem Assemblercode des Sprite-Editors. Wir werden hier also nach und nach die Bausteine kennenlernen, aus denen sich dann am Ende der vollständige Code des Sprite-Editors zusammensetzt.

Leiten wir zunächst her, warum wir diese Unterprogramme überhaupt brauchen und warum wir es mit 16bit Werten zu tun bekommen.

Wenn wir aus gegebener Zeile und Spalte am Bildschirm die entsprechende Adresse im Bildschirmspeicher berechnen wollen, dann errechnet sich diese Adresse aus der folgenden Formel:

$$\text{Zeile} * 40 + \text{Spalte} + 1024$$

Hier verlassen wir bereits bei der Multiplikation der Zeile mit 40 schon sehr bald den Wertebereich, der sich mit 8 Bits darstellen lässt. Der Wertebereich für den Zeilenwert reicht von 0 bis 24 und schon ab dem Zeilenwert 7 ergibt sich das Produkt 280 und dieser Wert lässt sich mit 8 Bits nicht mehr darstellen.

Der Wertebereich für den Spaltenwert reicht von 0 bis 39, dieser liegt also noch innerhalb des Wertebereichs, der sich mit 8 Bits darstellen lässt, aber der Wert 1024 liegt bereits wieder außerhalb.

So oder so, das Ergebnis wird selbst bei den kleinstmöglichen Werten für die Zeile und Spalte ein 16bit Wert sein, denn durch den Wert 1024 beträgt die kleinstmögliche Adresse

$$0 * 40 + 0 + 1024 = 1024$$

Beim Ausdruck „Zeile * 40“ stoßen wir bereits auf das erste Problem: Wie multipliziert man in Assembler eine Zahl mit 40?

Wie man Multiplikationen in Assembler umsetzt, haben wir bisher noch nicht gelernt. In diesem Buch werden wir das auch nicht lernen, weil ich allgemeine Routinen zur Multiplikation und Division erst im Buch für Fortgeschrittene vorstellen werde.

Wir haben jedoch bereits gelernt, wie man eine Zahl durch Anwendung des Befehls ASL mit einer Zweierpotenz multiplizieren kann.

Glücklicherweise kann man eine Multiplikation mit 40 leicht auf eine Kombination aus Multiplikationen mit Zweierpotenzen zurückführen.

Um also eine Zahl mit 40 zu multiplizieren, gehen wir folgendermaßen vor:

- Wir multiplizieren die Zahl zuerst mit 32 (also mit 2 hoch 5) und merken uns das Ergebnis.
- Wir multiplizieren die Zahl mit 8 (also mit 2 hoch 3) und addieren das Ergebnis zum Ergebnis der Multiplikation mit 32. Die Summe dieser beiden Produkte entspricht dann dem Produkt aus der Zahl und 40.
- Doch hier stoßen wir auf das nächste Problem: Wie addiert man zwei 16bit Zahlen? Wie man das macht, werden wir erfahren, nachdem wir unser Unterprogramm für die 16bit Version des Befehls ASL fertiggestellt haben.

Spielen wir das doch mal anhand eines Beispiels durch. Angenommen, wir haben in unserer Positionsangabe den Zeilenwert 18.

Nun müssen wir 18 mit 40 multiplizieren. Wenn wir nach den soeben genannten Schritten vorgehen, dann multiplizieren wir zuerst 18 mit 32 und dann 18 mit 8. Die Summe dieser beiden Produkte entspricht dann dem Produkt aus 18 und 40.

$$18 * 32 = 576$$

$$18 * 8 = 144$$

$$576 + 144 = 720$$

Rechnen wir nach, $18 * 40 = 720$, passt also!

Soweit so gut, dann machen wir uns mal an die Umsetzung der 16bit Version des ASL Befehls.

Wiederholen wir jedoch zunächst die Arbeitsweise des Befehls ASL. Angenommen im Akkumulator befindet sich der Wert 200 (binär %1100 1000 bzw. hexadezimal \$C8) und wir wollen auf diesen Wert den Befehl ASL anwenden.

Inhalt des Akkumulators vor und nach Ausführung des Befehls ASL:

	7	6	5	4	3	2	1	0
vor ASL	1	1	0	0	1	0	0	0
Nach ASL	1	0	0	1	0	0	0	0

Inhalt des Carryflags: 1

Was passiert nun beim Ausführen des Befehls ASL?

Auf der rechten Seite wandert ein Bit mit dem Inhalt 0 herein (dies ist das orange markierte Bit).

Dadurch werden alle anderen Bits um eine Position nach links verschoben und das grün markierte Bit fällt heraus und wandert in das Carryflag im Statusregister.

In diesem Fall hier hat das Carryflag nach der Ausführung des Befehls ASL den Inhalt 1, weil das grün markierte Bit den Inhalt 1 hat.

Ausgerüstet mit dem Wissen um die Funktionsweise des Befehls ASL, können wir uns nun an die Umsetzung der 16bit Version dieses Befehls machen.

Angenommen, wir wollen die Bits der 16bit Zahl 2500 um eine Position nach links verschieben. Mathematisch gesehen entspricht dies einer Multiplikation mit 2, das Ergebnis sollte also 5000 lauten.

Die binäre Darstellung der dezimalen Zahl 2500 lautet %0000 1001 1100 0100 oder \$09C4 in hexadezimaler Schreibweise.

Das höherwertige Byte lautet %0000 1001 bzw. \$09 und das niederwertige Byte %1100 0100 oder \$C4.

Doch wie führen wir diese Verschiebung durch? Ein erster Gedanke wäre, das höherwertige Byte und das niederwertige Byte jeweils um eine Bitposition nach links zu schieben und die beiden Werte dann wieder zusammenzusetzen.

Hört sich gut an, dann machen wir das mal.

Wenn wir das niederwertige Byte %1100 0100 um eine Bitposition nach links schieben, dann ergeben sich folgende Werte:

	7	6	5	4	3	2	1	0
vor ASL	1	1	0	0	0	1	0	0
nach ASL	1	0	0	0	1	0	0	0

Auf der rechten Seite ist ein Bit mit dem Inhalt 0 hereingewandert (orange markiert) und das grün markierte Bit fällt auf der linken Seite heraus und wandert ins Carryflag. In diesem Fall hat dieses nun den Inhalt 1, weil das Bit ganz links im niederwertigen Byte den Inhalt 1 hat.

Wenn wir das höherwertige Byte %0000 1001 um eine Bitposition nach links schieben, dann ergibt sich folgender Wert:

	7	6	5	4	3	2	1	0
vor ASL	0	0	0	0	1	0	0	1
nach ASL	0	0	0	1	0	0	1	0

Auf der rechten Seite ist ein Bit mit dem Inhalt 0 hereingewandert (orange markiert) und das grün markierte Bit fällt auf der linken Seite heraus und wandert ins Carryflag. In diesem Fall hat dieses nun den Inhalt 0, weil das Bit ganz links im höherwertigen Byte den Inhalt 0 hat.

Nun setzen wir diese beiden Ergebnisse zusammen und prüfen, ob wir auf dem richtigen Weg waren.

Niederwertiges Byte nach der Ausführung von ASL: %1000 1000

Höherwertiges Byte nach der Ausführung von ASL: %0001 0010

Zusammengesetzt ergibt das den 16bit Wert %0001 0010 1000 1000, hexadezimal \$1288 oder dezimal 4744. Passt also nicht, aber wo liegt der Fehler?

Das Problem ist, dass beim Verschieben des niederwertigen Bytes %1000 1000 das ganz links stehende Bit mit dem Inhalt 1 herausfällt. Dadurch fehlt es jedoch dann im zusammengesetzten Ergebnis und der resultierende Wert ist natürlich falsch.

Richtigerweise hätte das Bit, das im niederwertigen Byte durch das Verschieben herausfällt, in das ganz rechts liegende Bit des höherwertigen Bytes verschoben werden müssen.

Wenn das Bit, welches herausfällt, den Inhalt 0 hat, dann ist das kein Problem in Bezug auf das Endergebnis, aber im Falle des Inhalts 1 eben schon.

Doch wie lösen wir das Problem nun?

An dieser Stelle möchte ich Ihnen nun den Befehl ROL (rotate left) vorstellen. Dieser Befehl arbeitet ähnlich wie der Befehl ASL, nur mit dem Unterschied, dass auf der rechten Seite immer ein Bit mit dem Inhalt des Carryflags hereinwandert und nicht immer ein Bit mit dem Inhalt 0 so wie beim Befehl ASL.

Wie beim Befehl ASL fällt bei der Verschiebung der Bits auf der linken Seite ein Bit heraus, welches ins Carryflag wandert.

Hier zur Veranschaulichung ein Beispiel:

Nachfolgend der Inhalt des Akkumulators vor und nach Ausführung des Befehls ROL und angenommen, das Carryflag hat den Inhalt 0.

	7	6	5	4	3	2	1	0
vor ROL	1	1	0	0	1	0	0	0
nach ROL	1	0	0	1	0	0	0	0

Auf der rechten Seite ist ein Bit mit dem Inhalt 0 hereingewandert (orange markiertes Bit), da das Carryflag vor der Ausführung des Befehls ja den Inhalt 0 hatte. Nach der Ausführung des Befehls hat das Carryflag nun den Inhalt 1, denn das grün markierte Bit ist ins Carry Flag gewandert.

Führen wir den Befehl ROL nochmals aus, dann haben wir im Akkumulator vor und nach der Ausführung folgende Inhalte:

	7	6	5	4	3	2	1	0
vor ROL	1	0	0	1	0	0	0	0
nach ROL	0	0	1	0	0	0	0	1

Wie wir hier am orange markierten Bit sehen, ist nun auf der rechten Seite ein Bit mit dem Inhalt 1 hereingewandert, da das Carryflag durch den vorherigen Schritt den Inhalt 1 bekommen hatte.

Das grün markierte Bit ist wieder ins Carryflag gewandert, hier ebenfalls mit dem Inhalt 1.

Führt man den Befehl ROL nacheinander immer wieder aus, wandern die Bits über das Carryflag als Zwischenstation im Kreis (Bitrotation).

Möglicherweise fragen Sie sich jetzt, wozu so etwas gut sein soll. Ich muss zugeben, dass mir lange Zeit der Sinn und Zweck dieser Rotationsbefehle nicht klar war, doch das hat sich im Zuge der Umsetzung der 16bit Version des Befehls ASL geändert.

Wenn wir nun für die Bitverschiebung des höherwertigen Bytes nicht den Befehl ASL, sondern den Befehl ROL verwenden, dann bringt uns das genau jene Lösung, die wir brauchen.

Warum? Das Bit, welches uns vorhin verlorengegangen ist, haben wir ja noch im Carryflag zur Verfügung. Wenn wir nun auf das höherwertige Byte den Befehl ROL anwenden, dann wandert auf der rechten Seite genau dieses Bit herein und steht damit genau dort wo es sein soll, nämlich an der ersten Bitposition im höherwertigen Byte.

Die restlichen Bits werden wie bereits bekannt nach links verschoben und das Ergebnis ist nun korrekt.

Spielen wir das also mal durch:

Das Carryflag hat nach der Anwendung des Befehls ASL auf das niederwertige Byte den Inhalt 1 und der Inhalt des höherwertigen Bytes vor Anwendung des Befehl ROL lautet %0000 1001.

Wenn wir nun den Befehl ROL auf das höherwertige Byte anwenden, dann ergeben sich vor/nach Ausführung des Befehls folgende Inhalte:

	7	6	5	4	3	2	1	0
vor ROL	0	0	0	0	1	0	0	1
nach ROL	0	0	0	1	0	0	0	1

Das orange markierte Bit mit dem Inhalt 1 ist der hereingewanderte Inhalt des Carryflags.

Versuchen wir nun erneut, die Ergebnisse der beiden Verschiebungen zusammenzusetzen.

Niederwertiges Byte nach der Ausführung von ASL: %1000 1000

Höherwertiges Byte nach der Ausführung von ROL: %0001 0011

Zusammengesetzt ergibt das nun den korrekten 16bit Wert %0001 0011 1000 1000, hexadezimal \$1388 oder dezimal 5000.

Da uns die mathematische Vorgehensweise nun klar ist, können wir uns die konkrete Umsetzung in Assemblercode ansehen.

```
-----
;
; as116
; shiftet eine 16bit zahl um eine
; bestimmte anzahl an stellen
; nach links
;
; parameter:
; zahl: lo/hi in $fd/$fe
; stellen: x register
;
; rueckgabewerte:
; geshiftete zahl: lo/hi in $fd/$fe
;
; aendert:
; x,status
;
as116
as116_loop
    ; lo byte shiften
    asl $fd
    ; hi byte shiften
    rol $fe
    dex
    bne as116_loop
    rts
```

Hier sehen wir, dass durch den Befehl ASL \$FD das niederwertige Byte um eine Bitposition nach links verschoben wird. Im Anschluss geschieht dasselbe mit dem höherwertigen Byte durch den Befehl ROL \$FE, nur eben mit dem Unterschied, dass bei letzterem Befehl auf der rechten Seite nicht permanent ein Bit mit dem Inhalt 0 hereinwandert, sondern eines, welches dem aktuellen Inhalt des Carry Flags entspricht.

Das Unterprogramm bietet die Möglichkeit, eine 16bit Zahl nicht nur um eine, sondern um mehrere Stellen zu verschieben. Die Anzahl der Stellen übergeben wir als Parameter im X Register. Wir bilden also eine Schleife, die bei jedem Durchlauf die Bits der 16bit Zahl um eine weitere Position nach links verschiebt.

Jeder Durchlauf vermindert den Inhalt des X Registers um 1 und wenn wir schließlich bei 0 angelangt sind, wird die Schleife verlassen und wir haben die verschobene 16bit Zahl in den Speicherstellen \$FD (niederwertiges Byte) und \$FE (höherwertiges Byte) als Ergebnis zur Verfügung.

Mit diesem Unterprogramm können wir nun eine 16bit Zahl mit 2, 4, 8, 16, 32, 64, 128 usw. Multiplizieren.

Um nochmal auf den Befehl ROR zurückzukommen: Mit seiner Hilfe könnten sie eine 16bit Version des Befehls LSR bauen. Damit wären dann Divisionen durch Zweierpotenzen möglich.

Divisionen kommen in diesem Programm zwar nicht zur Anwendung, es wäre jedoch eventuell eine gute Übung.

Anmerkung: So wie es für den Befehl ASL einen Gegenspieler in Form des Befehls LSR gibt, existiert auch ein solcher für den Befehl ROL. Dieser nennt sich ROR, kommt aber beim Sprite-Editor nicht zur Anwendung.

Als nächstes werden wir uns mit der Addition zweier 16bit Zahlen beschäftigen, denn diese werden wir noch öfter benötigen.

Addition zweier 16bit Zahlen

Wir haben die Addition zweier 16bit Zahlen zwar bereits im Kapitel über Zahlensysteme angesprochen, aber ich möchte die Vorgehensweise trotzdem nochmals wiederholen, um sich alles wieder in Erinnerung zu rufen.

Zunächst jedoch ein paar wiederholende Worte zur 8bit Addition. Angenommen der Akkumulator enthält den dezimalen Wert 100 und wir addieren mit dem Befehl ADC den Wert 200 hinzu. Das Ergebnis 300 ist zu groß für den 8bit breiten Akkumulator. Es kommt also zu einem Überlauf und dies wird durch ein gesetztes Carryflag signalisiert. Liegt das Ergebnis einer Addition jedoch zwischen 0 und 255, so wird das Carryflag nach der Durchführung der Addition nicht gesetzt.

Vor der Durchführung einer Addition ist es daher wichtig, das Carryflag zurückzusetzen, da dieses sonst in die Addition miteinfließt.

Angenommen, wir wollen die Zahlen \$10 (dezimal 16) und \$20 (dezimal 32) addieren. Das Ergebnis würde \$30 (dezimal 48) lauten. Ist das Carryflag vor der Ausführung des Befehls ADC nicht gesetzt, dann erhalten wir auch genau dieses Ergebnis. Ist hingegen das Carryflag gesetzt, dann lautet das Ergebnis \$31 (dezimal 49), weil das Carryflag bei der Addition miteinbezogen wird.

Hier nochmals zur Veranschaulichung der Addition \$10 + \$20 und nicht gesetztem Carryflag:

	7	6	5	4	3	2	1	0
\$10	0	0	0	1	0	0	0	0
\$20	0	0	1	0	0	0	0	0
Carryflag								0
\$30	0	0	1	1	0	0	0	0

Und hier der Fall, dass das Carryflag gesetzt wäre:

	7	6	5	4	3	2	1	0
\$10	0	0	0	1	0	0	0	0
\$20	0	0	1	0	0	0	0	0
Carryflag								1

\$31	0	0	1	1	0	0	0	1
------	---	---	---	---	---	---	---	---

Im Kontext einer 8bit Addition vor der Ausführung also immer das Carryflag mit dem Befehl CLC zurücksetzen!

Kommen wir nun zur 16bit Addition. Die Vorgehensweise ist eigentlich recht simpel, hier eine Schritt für Schritt Anleitung:

- Zurücksetzen des Carryflags mit dem Befehl CLC, da vor Beginn der Addition ja noch kein Überlauf stattgefunden haben kann.
- Addieren der niederwertigen Bytes der beiden 16bit Zahlen und speichern des Ergebnisses im niederwertigen Byte der Summe. Falls es bei der Addition zu einem Überlauf kommt, wird dies durch ein gesetztes Carryflag angezeigt.
- Addieren der höherwertigen Bytes der beiden 16bit Zahlen und speichern des Ergebnisses im höherwertigen Byte der Summe. Falls es bei der Addition der niederwertigen Bytes zu einem Überlauf kam, fließt dieser in die Summe der höherwertigen Bytes mit einer Wertigkeit von 256 ein.

Hier sehen Sie den Assembler-Code zu dem Unterprogramm:

```

;-----
; adc16
; addiert zwei 16bit zahlen
;
; parameter:
; zahl1:  lo/hi  in  $fb/$fc
; zahl2:  lo/hi  im  x/y register
;
; rueckgabewerte:
; summe:  lo/hi  in  $fb/$fc
;
; aendert:
; a,status
;
adc16
    ; lo bytes addieren
    clc
    txa
    adc $fb
    sta $fb

    ; hi bytes addieren
    tya
    adc $fc
    sta $fc

    rts

```

Wie aus der Dokumentation hervorgeht, befindet sich die erste Zahl aufgeteilt auf die Speicherstellen \$FB / \$FC und die zweite Zahl aufgeteilt auf die Register X und Y. Die beiden niederwertigen Bytes befinden sich also in der Speicherstelle \$FB und im X Register, wohingegen sich die höherwertigen Bytes in der Speicherstelle \$FC und im Y Register befinden.

Nach der Durchführung der Addition soll die Summe aufgeteilt auf die Speicherstellen \$FB (niederwertiges Byte) und \$FC (höherwertiges Byte) verfügbar sein.

Durch den Befehl CLC wird hier zunächst das Carryflag gelöscht. Dann wird durch den Befehl TXA das niederwertige Byte der zweiten Zahl in den Akkumulator kopiert und durch den Befehl ADC \$FB das niederwertige Byte der ersten Zahl hinzuaddiert. Hier kann es wie gesagt zu einem Überlauf kommen, welcher durch ein gesetztes Carryflag signalisiert wird.

Im Akkumulator befindet sich nun die Summe der niederwertigen Bytes und diese wird durch den Befehl STA \$FB in das niederwertige Byte des Ergebnisses geschrieben.

Als nächstes wird durch den Befehl TYA das höherwertige Byte der zweiten Zahl in den Akkumulator kopiert und durch den Befehl ADC \$FC das höherwertige Byte der ersten Zahl hinzuaddiert. Falls es bei der Addition der niederwertigen Bytes zu einem Überlauf kam, das Carryflag also gesetzt wurde, fließt dieses in die Addition mit ein.

Im Akkumulator befindet sich nun die Summe der höherwertigen Bytes und diese wird durch den Befehl STA \$FC in das höherwertige Byte des Ergebnisses geschrieben.

Nun befindet sich die Summe mit dem niederwertigen Byte in der Speicherstelle \$FB und mit dem höherwertigen Byte in der Speicherstelle \$FC.

Durch die beiden Unterprogramme asl16 und adc16 haben wir nun alle Mittel in der Hand, um das eingangs erwähnte Problem zu lösen, nämlich die Umrechnung einer Position am Bildschirm in Form eines Zeilen- und eines Spaltenwertes in die entsprechende Adresse im Bildschirmspeicher.

Bevor wir uns nun den Assemblercode im Detail ansehen, möchte ich Ihnen kurz skizzieren, wie das Unterprogramm funktioniert.

Das Unterprogramm soll eine Position am Bildschirm, welche durch Zeile und Spalte gegeben ist, in die entsprechende Adresse im Bildschirmspeicher umrechnen.

Die Zeile wird im Y Register und die Spalte im X Register als Parameter an das Unterprogramm übergeben.

Der erste Schritt besteht darin, den Inhalt der Speicherstellen \$FB und \$FC sowie den Inhalt des X Registers und des Y Registers zu sichern.

Warum? Wie Sie später noch sehen werden, spielen die Speicherstellen \$FB und \$FC eine wichtige Rolle für den Sprite-Editor und deswegen müssen wir diese Inhalte sichern und vor der Rückkehr aus dem Unterprogramm wiederherstellen.

Das gilt auch für den Inhalt des X Registers und des Y Registers. Diese beiden Inhalte müssen wir deswegen sichern, weil sie durch nachfolgende Berechnungen verändert werden, aber nach Abschluss der Berechnung wieder mit ihrem ursprünglichen Inhalt gebraucht werden.

Im nächsten Schritt wollen wir die Zeile mit 40 multiplizieren. Wie bereits erwähnt, multiplizieren wir zuerst die Zeile mit 32, merken uns das Ergebnis, multiplizieren dann die Zeile mit 8 und addieren die beiden Produkte. Diese Summe entspricht dann dem Ergebnis der Multiplikation des Zeilenwertes mit 40.

Als nächstes addieren wir zu diesem Wert jenen Wert, den wir als Parameter für die Spalte übergeben haben und zuletzt müssen wir noch die Startadresse des Bildschirmspeichers, welche im

Normalfall der Adresse 1024 entspricht, hinzuaddieren. Dann haben wir endlich die gewünschte Adresse im Bildschirmspeicher.

Nachdem das Unterprogramm seine Aufgabe erfüllt hat, legt es das niederwertige Byte dieser Adresse im X Register und das höherwertige Byte im Y Register ab.

Die Adresse ließe sich dann mit der Formel

$$\text{Inhalt des X Registers} + 256 * \text{Inhalt des Y Registers}$$

berechnen.

Kommen wir nun zum Assemblercode des Unterprogramms. Ich habe ihn mit vielen Kommentaren versehen, aber es folgt im Anschluss an den Assemblercode trotzdem noch eine detaillierte Erklärung.

```
-----  
: calcposaddr  
: berechnet die adresse einer  
: position zeile/spalte im  
: bildschirmspeicher  
:  
: parameter:  
: spalte: x register  
: zeile: y register  
:  
: rueckgabewerte:  
: adresse: lo/hi im x/y register  
:  
: aendert:  
: a,x,y,status,$fb,$fc  
:  
calcposaddr  
: speicherstellen $fb und $fc  
: auf dem stack sichern, da  
: sie in diesem unterprogramm  
: ueberschrieben werden  
:  
lda $fb  
pha  
:  
lda $fc  
pha  
:  
: auch das x register und das  
: y register (welche die  
: parameter fuer zeile und  
: spalte beinhalten) auf dem  
: stack sichern, weil sie  
: durch die nachfolgenden  
: berechnungen ueberschrieben  
: werden  
:  
txa  
pha  
:  
tya  
pha  
:  
: zeile * 32 berechnen  
: lo (zeile) nach $fd  
sty $fd  
:  
: hi (zeile) nach $fe  
: ist immer $00 weil der  
: wert fuer die zeile nur von
```

```

; 0 bis 24 reicht
ldy #$00
sty $fe

; wir wollen mit 32, also
; mit 2 hoch 5 multiplizieren,
; daher 5 stellen als
; parameter im x register
; uebergeben

ldx #$05
jsr asl16

; ergebnis in $fb/$fc
; zwischenspeichern

lda $fd
sta $fb

lda $fe
sta $fc

; zeile wieder vom stack
; holen
pla

; zeile * 8 berechnen
; lo (zeile) wieder nach $fd
tay
sty $fd

; hi (zeile) wieder nach $fe
ldy #$00
sty $fe

; wir wollen mit 8, also mit
; 2 hoch 3 multiplizieren,
; daher 3 stellen als
; parameter im x register
; uebergeben

ldx #$03
jsr asl16

; nun addieren wir
; zeile * 32 und zeile * 8

; zeile * 32 befindet sich
; bereits in $fb/$fc

; zeile * 8 (hier in $fd/$fe)
; fuer die addition nach
; x und y kopieren

ldx $fd
ldy $fe

jsr adc16

; spalte wieder vom stack
; holen
pla

; spalte ins x register holen
; dort steht dann lo (spalte)
tax

; hi (spalte) ist immer $00,
; da der wert fuer die spalte
; nur von 0 bis 39 reicht

```

```

ldy #$00
; nun addieren wir die spalte
; hinzu
; zeile * 40 ist in $fb/$fc
; spalte ist im
; x register und y register
jsr adc16
; nun addieren wir noch
; die startadresse des
; bildschirmspeichers hinzu
; diese adresse lautet im
; normalfall 1024 ($0400)
; lo ($0400) = $00 fuer die
; addition ins x register
ldx #$00
; hi ($0400) = $04 fuer die
; addition ins y register
ldy #$04
jsr adc16
; das ergebnis ist nun
; die gewuenschte adresse
; im bildschirmspeicher
; diese stellen wir im
; x register (lo) und
; y register (hi) zur
; verfuegung
ldx $fb
ldy $fc
; inhalte der speicherstellen
; $fb und $fc wiederherstellen
pla
sta $fc
pla
sta $fb
rts

```

Wir starten mit dem Abschnitt, welcher durch das Kommentar

```

; zeile * 32 berechnen

```

eingeleitet wird.

Das Unterprogramm asl16 erwartet die Zahl, welche bitweise verschoben werden soll, in den Speicherstellen \$FD (niederwertiges Byte) und \$FE (höherwertiges Byte).

Der Zeilenwert wurde als Parameter im Y Register abgelegt, daher wird mit dem Befehl STY \$FD das niederwertige Byte des Zeilenwertes in die Speicherstelle \$FD geschrieben.

Das höherwertige Byte des Zeilenwertes ist immer \$00, weil der Zeilenwert ja nur zwischen 0 und 24 liegen kann. Daher wird hier zuerst das Y Register mit dem Wert \$00 geladen und dieser dann mit dem Befehl STY \$FE in die Speicherstelle \$FE geschrieben.

Nun müssen wir noch im X Register die Anzahl der Stellen angeben, um die wir die Zahl verschieben wollen. In diesem Fall sind es 5 Stellen, denn wir wollen den Zeilenwert ja mit 32 multiplizieren, was einer Verschiebung von 5 Bitpositionen nach links entspricht.

Nach dem Aufruf von asl16 finden wir das Ergebnis, also das Produkt aus Zeilenwert und 32, mit dem niederwertigen Byte in der Speicherstelle \$FD und dem höherwertigen Byte in der Speicherstelle \$FE vor.

Dieses Produkt müssen wir uns irgendwo merken, weil wir die Speicherstellen \$FD und \$FE für die nächste Multiplikation brauchen. In diesem Fall merken wir uns das Produkt in den Speicherstellen \$FB und \$FC. Die Erklärung, warum ich mich ausgerechnet für diese beiden Speicherstellen entschieden habe, folgt in Kürze.

Als Nächstes steht die Multiplikation des Zeilenwertes mit 8 am Programm. Dieser Abschnitt wird durch das Kommentar

```
; zeile * 8 berechnen
```

eingeleitet.

Zu Beginn des Unterprogramms haben wir neben den Inhalten der Speicherstellen \$FB und \$FC auch die übergebenen Parameter für den Zeilen- und Spaltenwert auf dem Stack gesichert. Der Zeilenwert wurde als letzter Wert gesichert, d.h. er liegt an der obersten Stelle des Stacks und wir können ihn daher mit dem Befehl PLA direkt in den Akkumulator holen, um ihn dann wie vorhin bei der Multiplikation mit 32 in die Speicherstelle \$FD zu schreiben.

Da fällt mir gerade auf, dass ich mir im direkt auf das Kommentar folgenden Abschnitt

```
; lo (zeile) wieder nach $fd  
tay  
sty $fd
```

den Befehl TAY hätte sparen können und dass der Befehl STA \$FD gereicht hätte. Aber egal, lassen wir es so, denn falsch ist es ja nicht.

Auch hier müssen wir wieder den Wert \$00 in die Speicherstelle \$FE schreiben, da wie bereits erwähnt, das höherwertige Byte ja immer den Wert \$00 hat.

Da wir diesmal eine Multiplikation mit 8 durchführen wollen und dies einer Verschiebung um 3 Bitpositionen nach links entspricht, müssen wir das X Register mit dem Wert 3 laden.

Nach dem Aufruf von asl16 finden wir das Ergebnis wieder in den Speicherstellen \$FD und \$FE vor.

Nun müssen wir die beiden Produkte $zeile * 32$ und $zeile * 8$ addieren. Das Unterprogramm adc16 erwartet die erste Zahl verteilt auf die Speicherstellen \$FB und \$FC sowie die zweite Zahl verteilt auf das X Register und das Y Register.

Und jetzt kommt die Antwort auf die Frage, warum ich mich vorhin für die Zwischenspeicherung des Produkts des Zeilenwertes mit 32 für die Speicherstellen \$FB und \$FC entschieden habe.

Durch den Umstand, dass das Produkt $zeile * 32$ bereits wie von `adc16` erwartet, in den Speicherstellen `$FB` und `$FC` vorliegt, brauchen wir es nicht extra dorthin transportieren, sondern wir müssen nur das niederwertige Byte des zweiten Produkts $zeile * 8$ im X Register und das höherwertige Byte im Y Register bereitstellen.

Dies erfolgt über die Befehle `LDX $FD` und `LDY $FE`.

Nach dem Aufruf von `adc16` steht die Summe aus $zeile * 32$ und $zeile * 8$, also $zeile * 40$ aufgeteilt auf die Speicherstellen `$FB` und `$FC` zur Verfügung.

Nun müssen wir zu diesem Wert den Spaltenwert addieren. Diesen haben wir zu Beginn des Unterprogramms auf dem Stack gesichert und nachdem wir bereits den Zeilenwert vom Stack geholt haben, liegt nun der Spaltenwert auf der obersten Stelle des Stacks.

Wir holen ihn mit dem Befehl `PLA` von dort direkt in den Akkumulator und transportieren ihn von dort ins X Register, da das Unterprogramm `adc16` das niederwertige Byte der zweiten Zahl dort erwartet. Das höherwertige Byte der zweiten Zahl wird im Y Register erwartet und da dieses immer den Wert `$00` hat, brauchen wir nur das Y Register mit dem Wert `$00` zu laden.

Wiederum machen wir uns den Umstand zunutze, dass die Summe der beiden Produkte noch immer in den Speicherstellen `$FB` und `$FC` gespeichert ist und können ohne weitere Vorbereitungen umgehend den Aufruf von `adc16` starten.

Die Summe aus Zeile $* 40$ + Spalte steht uns dann wiederum aufgeteilt auf die Speicherstellen `$FB` und `$FC` zur Verfügung.

Nun müssen wir noch als letzten Schritt den Wert 1024 (`$0400`) hinzuaddieren. Dies ist ja im Normalfall die Startadresse des Bildschirmspeichers. Auch dies ist wieder ganz einfach, denn das letzte Zwischenergebnis ($Zeile * 40 + Spalte$) ist ja noch in den Speicherstellen `$FB` und `$FC` gespeichert.

Wir müssen also nur mehr das niederwertige Byte `$00` in das X Register und das höherwertige Byte `$04` in das Y Register laden. Nun noch ein letztesmal `adc16` aufgerufen und schon haben wir wieder in den Speicherstellen `$FB` und `$FC` das Ergebnis der Addition, welches diesesmal unserem Endergebnis entspricht, also der gewünschten Adresse im Bildschirmspeicher.

Sie sehen also, dass ich mir die Speicherstellen `$FB` und `$FC` zur Zwischenspeicherung des ersten Produkts nicht zufällig ausgesucht habe, sondern weil dadurch die Aufrufe des Unterprogramms `adc16` effektiver gestaltet werden können, weil sich ein Operand bereits dort befindet, wo er vom Unterprogramm erwartet wird und man sich nur mehr um die Übergabe des zweiten Operanden im X Register und Y Register kümmern muss.

Das Unterprogramm `adc16` legt das Ergebnis ebenfalls in den Speicherstellen `$FB` und `$FC` ab und so kann ein Berechnungsschritt direkt auf dem vorherigen aufbauen.

Abschließend müssen wir noch die Inhalte der Speicherstellen `$FB` und `$FC` in das X Register bzw. das Y Register kopieren und anschließend die ursprünglichen Inhalte der Speicherstellen `$FB` und `$FC` wiederherstellen, da wie bereits erwähnt, diese für den Sprite-Editor eine besondere Bedeutung haben.

Um die Unterprogramme asl16, adc16 und calcpasaddr im Zusammenspiel zu sehen, habe ich das Programm CALCADDR auf Diskette gespeichert.

Es enthält außer den genannten drei Unterprogrammen einen Hauptteil, welcher die Parameter setzt und das Unterprogramm calcpasaddr aufruft.

```
-----  
; hauptprogramm  
; hier wird das programm gestartet  
; start von basic aus mit sys 12288  
  
    *= $3000  
    ; spalte = 39 ($27)  
    ldx #$27  
    ; zeile = 24 ($18)  
    ldy #$18  
  
    ; adresse im  
    ; bildschirmspeicher berechnen  
    ; diese befindet sich nach  
    ; der beendigung des programms  
    ; im x register (lo byte) und  
    ; im y register (hi byte)  
  
    jsr calcpasaddr  
  
    rts
```

Hier wird anhand der Position, welche durch die Zeile 24 und Spalte 39 gegeben ist, demonstriert, wie diese in die entsprechende Adresse im Bildschirmspeicher umgerechnet wird.

Der Spaltenwert 39 (\$27) wird in das X Register und der Zeilenwert 24 (\$18) in das Y Register geladen. Dann wird das Unterprogramm calcpasaddr aufgerufen und wenn man nach der Beendigung des Programms wieder ins Basic zurückgekehrt ist, kann man die berechnete Adresse mit dem Befehl PRINT PEEK(781)+256*PEEK(782) ausgeben lassen:

```
READY.  
SYS 12288  
  
READY.  
PRINT PEEK(781)+256*PEEK(782)  
2023  
  
READY.
```

Rechnen wir abschließend nochmal anhand der einzelnen Schritte nach:

- $zeile * 32 = 24 * 32 = 768$
- $zeile * 8 = 24 * 8 = 192$
- $768 + 192 = 960$
- $960 + spalte = 960 + 39 = 999$
- $1024 + 999 = 2023$

Passt also!

So, nun haben wir für's Erste aber genug gerechnet und wenden uns der Darstellung der Benutzeroberfläche des Sprite-Editors zu.

Wenn wir den Sprite-Editor starten, dann ist folgendes Bild zu sehen:



Wie wir bereits wissen, bestehen die Spritedaten aus 21 Reihen zu je 3 Bytes. Der String am oberen Rand stellt die Bitpositionen innerhalb dieser drei Bytes dar.

Jede Reihe des Editorbereichs wird durch einen String dargestellt, der zu Beginn zwei Ziffern enthält. Diese stellen die Nummer der jeweiligen Reihe dar.

Der Rest des Strings besteht aus 24 Punkten, welche die Bitpositionen innerhalb der drei Bytes repräsentieren.

Wie sie im folgenden Ausschnitt des Editors sehen können, steht ein kleiner Punkt für eine „0“ und ein großer Punkt für eine „1“.



Der Cursor, welcher aussieht wie der gewohnte blinkende C64 Cursor, ist in der oberen linken Ecke zu sehen.

Dieser Cursor sieht zwar aus wie der C64 Cursor, er ist jedoch, wie bereits eingangs erwähnt, völlig unabhängig von diesem und zeigt sich eigentlich nur durch die reverse Darstellung des Zeichens, an der Position, an welcher sich der Cursor gerade befindet. Doch dazu später mehr, wenn wir die Steuerung des Cursors in Angriff nehmen.

Am unteren Rand sieht man einen String, welcher auf die Verfügbarkeit von hilfreichen Informationen hinweist, die durch Drücken der Tastenkombination SHIFT + H angezeigt werden können.

Die Benutzeroberfläche des Sprite-Editors ist also aus einzelnen Strings zusammengesetzt und daher brauchen wir nun ein Unterprogramm, mit dem wir einen beliebigen String an einer bestimmten Bildschirmposition ausgeben können.

Dieses möchte ich Ihnen nun ohne Umschweife vorstellen.

```

:-----
: printstr
: gibt einen null-terminierten string
: an der aktuellen cursorposition
: aus
:
: parameter:
: adresse des strings: lo/hi in $fd/fe
: spalte: y register
: zeile: x register
:
: rueckgabewerte:
: keine
:
: aendert:
: a,y,status
:
printstr
    ; cursor positionieren
    clc
    jsr $fff0
    ; string ausgeben
    ldy #$00
printstr_loop
    lda ($fd),y
    beq printstr_end
    jsr $ffd2
    iny
    jmp printstr_loop
printstr_end
    rts

```

Im Beschreibungsteil am Anfang begegnet uns so einiges an Neuem. Da wäre beispielsweise der Begriff „null-terminierter string“.

Was verbirgt sich dahinter?

Eigentlich nichts besonderes, ein null-terminierter String ist ein String, welcher am Ende ein Nullbyte enthält. Dieses gehört nicht zum Inhalt des Strings, sondern dient dazu, das Ende des Strings zu markieren.

Dem Unterprogramm werden drei Parameter übergeben:

- Die Adresse des Strings, wobei das niederwertige Byte der Adresse in der Speicherstelle \$FD und das höherwertige Byte der Adresse in der Speicherstelle \$FE stehen muss.
- Die Spalte an der der String angezeigt werden soll, diese muss im Y Register stehen.
- Die Zeile in der der String angezeigt werden soll, diese muss im X Register stehen.

Ein Ergebnis in Form eines Rückgabewertes, wie es beispielsweise vom Unterprogramm calcpaddr produziert wird, liefert uns das Unterprogramm nicht. Es gibt einen String am Bildschirm aus, nicht mehr und nicht weniger.

Innerhalb des Unterprogramms wird der Akkumulator, das Y Register und das Statusregister verändert.

Kommen wir nun zur Funktion des Unterprogramms.

Als Erstes wird der Cursor auf jene Position bewegt, welche wir als Parameter im X Register und Y Register übergeben haben. In den folgenden Ausführungen meine ich wirklich den C64 Cursor, denn durch seine Position wird bestimmt, wo die nächste Ausgabe auf dem Bildschirm stattfindet.

Für die Positionierung des Cursors wird hier die Kernal-Funktion PLOT verwendet, welche über die Adresse \$FFF0 aufgerufen werden kann. Sie erwartet die Zeile im X Register und die Spalte im Y Register. Der Inhalt des Carryflags entscheidet darüber, ob die Cursorposition ausgelesen oder eingestellt werden soll.

Wollen wir die Cursorposition auslesen, dann müssen wir das Carryflag vor dem Aufruf der Funktion durch den Befehl SEC setzen und wenn wir die Cursorposition einstellen wollen, dann müssen wir das Carryflag vor dem Aufruf der Funktion mit dem Befehl CLC löschen.

Genau dies ist hier der Fall. Wir wollen den Cursor auf eine bestimmte Position setzen und deswegen wird hier vor dem Aufruf der Funktion das Carryflag durch den Befehl CLC gelöscht.

Die Werte für Zeile und Spalte befinden sich ja bereits im X Register bzw. Y Register und daher können wir die Funktion PLOT durch den Befehl JSR \$FFF0 aufrufen.

Nun können wir uns an die Ausgabe des Strings machen. Über die Kernalfunktion CHROUT, welche über die Adresse \$FFD2 aufgerufen werden kann, werden wir den String Zeichen für Zeichen ausgeben, bis wir auf ein Nullbyte stoßen. Dieses markiert ja wie bereits erwähnt das Ende des Strings.

Soweit zur grundsätzlichen Vorgangsweise, doch soweit sind wir noch nicht. Ich muss Ihnen zuerst erklären, was es mit dem Ausdruck (\$FD),Y hinter dem Befehl LDA auf sich hat.

Sehen wir uns zunächst den Ausdruck (\$FD) an, auf die Bedeutung des Kommas gefolgt von dem Y kommen wir dann im Anschluss zu sprechen.

Wir lernen hier zusätzlich zu den vielen Adressierungsarten, welche wir bereits kennengelernt haben, noch eine weitere kennen. Diese hat den furchtbar kompliziert klingenden Namen „Indirekte Y-nachindizierte Zeropage Adressierung“.

Über diese Art der Adressierung haben wir die Möglichkeit, auf eine Speicherstelle zuzugreifen, deren Adresse in zwei aufeinanderfolgenden Speicherstellen innerhalb der Zeropage zu finden ist. Das niederwertige Byte und das höherwertige Byte dieser Adresse liegen also in direkt aufeinanderfolgenden Speicherstellen innerhalb der Zeropage. Die Anzahl der dafür nutzbaren Speicherstellen in der Zeropage ist sehr begrenzt.

Im Grunde beschränken sie sich auf die uns bereits bekannten Speicherstellen \$FB, \$FC, \$FD und \$FE, welche wir schon oft für diverse Zwecke verwendet haben. Die Speicherstellen, an der die Adresse in der Zeropage zu finden ist, wird durch die Adresse in der Klammer angegeben.

Durch den Ausdruck (\$FD) wird also festgelegt, dass sich das niederwertige Byte der Adresse in der Speicherstelle \$FD und das höherwertige Byte der Adresse in der Speicherstelle \$FE befindet.

Da die beiden Speicherstellen ja direkt aufeinanderfolgen, ist es nicht nötig, beide Speicherstellen anzugeben, sondern es reicht die Angabe der ersten Speicherstelle, also jener Speicherstelle in der wir das niederwertige Byte der Adresse hinterlegt haben.

Man kann sich das folgendermaßen vorstellen:

Angenommen, wir möchten einen Brief versenden, wobei wir die Adresse vorerst noch nicht kennen.

Wir wissen jedoch, dass diese Adresse verteilt auf zwei Zettel in Postfächern mit direkt aufeinander folgenden Nummern hinterlegt ist, wobei uns die Nummer des ersten Postfachs bekannt ist.

Auf dem ersten Zettel steht die Postleitzahl und der Ort, auf dem zweiten Zettel die Straße und Hausnummer.

Wir öffnen also das erste Postfach, entnehmen den Zettel, öffnen das zweite Postfach, entnehmen auch diesen Zettel und setzen die Informationen auf beiden Zetteln zu einer Adresse zusammen, an die wir den Brief nun versenden können.

Angewandt auf das obige Beispiel würde das erste Postfach die Nummer \$FD und das zweite Postfach die Nummer \$FE haben. Auf dem Zettel aus dem ersten Postfach steht das niederwertige

Byte der Adresse (Postleitzahl und Ort) und auf dem Zettel aus dem zweiten Postfach steht das höherwertige Byte (Straße und Hausnummer).

Zusammengesetzt ergeben diese Informationen dann die gewünschte Adresse.

Ich will Ihnen nun an einem ganz einfachen Beispiel demonstrieren, wie diese Art der Adressierung funktioniert.

Angenommen, wir wollen in die linke obere Ecke des Bildschirms ein A schreiben, wobei wir direkt in den Bildschirmspeicher schreiben wollen.

Die Adresse der linken oberen Ecke des Bildschirms stellt gleichzeitig die Anfangsadresse des Bildschirmspeichers dar, welche im Normalfall 1024 (\$0400) lautet.

Ich habe ein kleines Programm namens YINDADDR vorbereitet, das Ihnen die Funktionsweise der indirekten y nachindizierten Zeropage Adressierung demonstrieren soll.

```
;-----  
; hauptprogramm  
; hier wird das programm gestartet  
; start von basic aus mit sys 12288  
  
    *= $3000  
  
    ; grosses a in die linke  
    ; obere ecke des bildschirms  
    ; schreiben  
  
    ; die adresse im  
    ; bildschirmspeicher lautet  
    ; $0400 (1024)  
  
    ; screencode fuer grosses a  
    lda #$01  
  
    ; und in den  
    ; bildschirmspeicher schreiben  
    sta $0400  
  
    ; und nun ueber die indirekte  
    ; y nachindizierte  
    ; zeropage adressierung  
  
    ; diesesmal wollen wir ein  
    ; grosses b rechts neben dem  
    ; grossen a ausgeben  
  
    ; die adresse im  
    ; bildschirmspeicher lautet  
    ; $0401 (1025)  
  
    ; niederwertiges byte der  
    ; adresse in die  
    ; speicherstelle $fb schreiben  
    lda #$01  
    sta $fb  
  
    ; hoeherwertiges byte der  
    ; adresse in die  
    ; speicherstelle $fc schreiben  
    lda #$04  
    sta $fc
```

```

; index = 0
ldy #$00
; screencode fuer grosses b
lda #$02
; diesen wert in die
; speicherstelle schreiben
; welche durch die
; speicherstellen $fb und $fc
; sowie das y register
; festgelegt ist
sta ($fb),y
rts

```

Im Programm wird hier zunächst ein A in der linken oberen Ecke des Bildschirms ausgegeben.

Dazu wird zunächst der Screencode dieses Zeichens (also \$01) in den Akkumulator geladen und von dort in die Speicherstelle \$0400 (1024) geschrieben. Ist also nichts neues, haben wir schon oft gemacht.

Als nächstes wird nun ein B rechts neben dem A ausgegeben. Das werden wir diesmal jedoch nicht über die absolute Adressierung, also durch direkte Angabe der Speicheradresse hinter dem Befehl STA lösen, sondern eben über die indirekte Y nachindizierte Zeropage Adressierung.

Das B soll rechts neben dem A ausgegeben werden, der Screencode des Zeichens B muss also an die Adresse \$0401 (1025) geschrieben werden.

Dazu schreiben wir zunächst das niederwertige Byte der Adresse (\$01) in die Speicherstelle \$FB und das höherwertige Byte der Adresse (\$04) in die Speicherstelle \$FC.

Ich habe hier zur Abwechslung mal andere Speicherstellen genutzt, nämlich die Speicherstellen \$FB und \$FC. Ich hätte genauso gut die Speicherstellen \$FC und \$FD oder \$FD und \$FE nutzen können.

Als nächstes laden wir das Y Register mit dem Wert \$00, dieser dient nachfolgend bei der Adressierung als zusätzlicher Index, welcher in die Bildung der Zieladresse einfließt.

Der Befehl LDA #\$02 bringt den Screencode des Zeichens B in den Akkumulator, welchen wir nur mehr in den Bildschirmspeicher bringen müssen.

Nun kommt der große Moment in Bezug auf die Bildung der Zieladresse im Bildschirmspeicher.

In der Speicherstelle \$FB steht nun das niederwertige Byte der Adresse und in der Speicherstelle \$FC das höherwertige Byte.

Die CPU bildet die Adresse dann durch die Formel

$$\text{Inhalt der Speicherstelle \$FB} + 256 * \text{Inhalt der Speicherstelle \$FC}$$

und zählt noch als Index den Inhalt des Y Registers hinzu (daher das Komma gefolgt von Y).

Somit ergibt sich als Zieladresse $\$01 + 256 * \$04 + \$00 = 1 + 1024 + 0 = 1025 (\$0401)$

An diese Adresse wird nun der Screencode des Zeichens B geschrieben, d.h. das B wird wie gewünscht neben dem A ausgegeben.

Soweit so gut, aber was war denn nun der große Vorteil gegenüber der absoluten Adressierung, welche wir bei der Ausgabe des A verwendet haben? Immerhin haben wir da um einige Befehle mehr benötigt.

Der Vorteil liegt darin, dass die Zieladresse aus den zwei Speicherstellen \$FB und \$FC gelesen wird, deren Inhalt natürlich jederzeit geändert werden kann. Als zusätzlicher, veränderlicher Faktor kommt noch der Inhalt des Y Registers hinzu, in dem man einen Index angeben kann, der noch zur Zieladresse hinzuaddiert wird.

Es wäre in diesem Beispiel auch möglich gewesen, die Zieladresse \$0401 (1025) auf andere Art und Weise zu bilden. Wir hätten als Zieladresse die Adresse \$0400 (1024) wählen und diese auf die Speicherstellen \$FB und \$FC verteilen können.

Durch Angabe des Index \$01 im Y Register wären wir dann auf dieselbe Adresse gekommen.

Dann hätte sich die Zieladresse aus

$\$00$ (Inhalt der Speicherstelle \$FB) + $256 * \$04$ (Inhalt der Speicherstelle \$FC) + $\$01$ (Index im Y Register) = $0 + 1024 + 1 = 1025 (\$0401)$

errechnet.

Bei der absoluten Adressierung hingegen, die wir beim Befehl STA \$0400 zur Ausgabe des Zeichens A verwendet haben, sind wir auf die Speicheradresse \$0400 festgelegt.

Doch wie wird nun diese Art der Zeropage-Adressierung innerhalb des Unterprogramms printstr genutzt?

Dazu habe ich das Program PRINTSTR erstellt, in dem das Unterprogramm printstr zur Anwendung kommt.

Es werden zwei Strings an unterschiedlichen Positionen auf dem Bildschirm ausgegeben und um den Bezug zum Sprite-Editor aufrecht zu erhalten, habe ich dafür den String am oberen Rand, welcher die Bitpositionen darstellt, sowie den String am unteren Rand, welcher den Hinweis auf die verfügbaren Hilfsinformationen darstellt, ausgewählt.

Diese beiden Strings sind am Ende des Programms im Datenabschnitt abgelegt.

```
;-----  
;  daten  
  
bitposstr  
    .text  "7654321076543210"  
    .null  "76543210"  
  
helpinfo  
    .text  "press shift + h for "  
    .null  "help"
```


Der Grund, warum ich die Strings aufteilen musste, besteht in der begrenzten Ausgabebreite im TMP. Diese Aufteilung auf zwei Zeilen lässt die Strings zunächst nicht wie eine Einheit erscheinen, aber im Speicher liegen die Strings „7654321076543210“ und „76543210“ direkt hintereinander, wobei hinter letzterem noch automatisch ein Nullbyte angehängt wird, da die Definition mit .null erfolgt ist.

Erfolgt die String-Definition durch .text, dann wird am Ende kein Nullbyte ergänzt. Man kann dieses aber natürlich jederzeit manuell durch eine Zeile mit dem Inhalt .byte \$00 hinzufügen.

Dieses Nullbyte am Ende der Strings ist wichtig für die Ausgabe durch das Unterprogramm printstr, da es als Markierung für das Ende des Strings dient. Wenn das abschließende Nullbyte fehlt, dann würde das Unterprogramm printstr solange Zeichen ausgeben, bis es im Speicher zufällig auf ein Nullbyte trifft.

```
-----  
; hauptprogramm  
; hier wird das programm gestartet  
; start von basic aus mit sys 12288  
  
    *= $3000  
  
    ; string mit bitpositionen  
    ; ausgeben  
  
    ; niederwertiges byte der  
    ; adresse von bitposstr  
    ; in speicherstelle $fd  
    ; schreiben  
    lda #<bitposstr  
    sta $fd  
  
    ; hoeherwertiges byte der  
    ; adresse von bitposstr  
    ; in speicherstelle $fe  
    ; schreiben  
    lda #>bitposstr  
    sta $fe  
  
    ; zeile=0, spalte=9  
    ldx #$00  
    ldy #$09  
  
    ; bitposstr ausgeben  
    jsr printstr  
  
    ; string mit hinweis auf  
    ; verfuegbare  
    ; hilfeinformationen ausgeben  
  
    ; niederwertiges byte der  
    ; adresse von helpinfo  
    ; in speicherstelle $fd  
    ; schreiben  
    lda #<helpinfo  
    sta $fd  
  
    ; hoeherwertiges byte der  
    ; adresse von helpinfo  
    ; in speicherstelle $fe  
    ; schreiben  
    lda #>helpinfo  
    sta $fe  
  
    ; zeile=22 ($16), spalte=8
```

```

        ldx #$16
        ldy #$08

        ; helpinfo ausgeben

        jsr printstr

        ; cursor in die linke obere
        ; ecke versetzen, damit die
        ; ausgabe nicht nach oben
        ; gescrollt wird

        clc
        ldx #$00
        ldy #$00
        jsr $fff0

        rts

;-----
; printstr
; gibt einen null-terminierten string
; an der aktuellen cursorposition
; aus
;
; parameter:
; adresse des strings: lo/hi in $fd/$fe
; spalte: y register
; zeile: x register
;
; rueckgabewerte:
; keine
;
; aendert:
; a,y,status
printstr
        ; cursor positionieren

        clc
        jsr $fff0

        ; string ausgeben

        ldy #$00

printstr_loop
        lda ($fd),y
        beq printstr_end
        jsr $ffd2
        iny
        jmp printstr_loop

printstr_end
        rts

;-----
; daten

bitposstr
        .text "7654321076543210"
        .null "76543210"

helpinfo
        .text "press shift + h for "
        .null "help"

```

Wenn wir den Assemblercode von oben beginnend durchsehen, fällt als erste Neuigkeit die Zeichenfolge „#<“ im Befehl LDA auf.

```
lda #<bitposstr
```

Einige Zeilen darunter findet man denselben Befehl, nur dass diesmal die Zeichenfolge „#>“ zu sehen ist.

```
lda #>bitposstr
```

Was hat es damit auf sich?

In unserem Assembler-Code sind mehrere Labels zu finden:

- printstr
- printstr_loop
- printstr_end
- bitposstr
- helpinfo

Diese Labels sind nichts anderes als Namen für Speicheradressen. Als wir noch im SMON programmiert haben, mussten wir in allen Befehlen die Speicheradressen tatsächlich über ihre Nummer ansprechen.

Hier nochmal eine Erinnerung an unsere Zeit mit SMON:

Address	Hex	Hex	Hex	Instruction
,1521	D0	F0		BNE 1513
,1523	60			RTS

.A 1523				
1523	A2	00		LDX #00
1525	BD	00	15	LDA 1500,X
1528	9D	00	30	STA 3000,X
152B	E8			INX
152C	E0	23		CPX #23
152E	D0	F5		BNE 1525
1530	A9	60		LDA #60
1532	8D	23	30	STA 3023
1535	60			RTS

1536	F			
,1523	A2	00		LDX #00
,1525	BD	00	15	LDA 1500,X
,1528	9D	00	30	STA 3000,X
,152B	E8			INX
,152C	E0	23		CPX #23
,152E	D0	F5		BNE 1525
,1530	A9	60		LDA #60
,1532	8D	23	30	STA 3023
,1535	60			RTS

. ■				

An der Adresse \$152E steht beispielsweise der Befehl BNE 1525. Die Zahl 1525 hinter dem Befehl BNE steht für die Speicheradresse \$1525.

Die Arbeit mit konkreten Speicheradressen wurde mit der Zeit natürlich sehr schwierig, weil man sich diese Zahlen nicht gut merken kann. Noch schwieriger wurde dies, wenn es Änderungen am Programm gab und sich die Speicheradressen verschoben haben.

Auf diese Art und Weise war kein vernünftiges Programmieren möglich und daher sind wir ja auf den TMP umgestiegen, da dieser neben seinen vielen anderen Vorteilen auch die Möglichkeit bietet, sogenannte Labels zu verwenden.

Da wir diese Labels beliebig benennen können, merken wir sie uns natürlich auch leichter.

Doch zurück zu den Zeichenfolgen #< und #>.

Die Zeichenfolge #< steht für das niederwertige Byte der Speicheradresse, welche durch das darauf folgende Label bezeichnet wird.

```
lda #<bitposstr
```

Hier steht die Zeichenfolge #< für das niederwertige Byte der Speicheradresse, die sich hinter dem Label bitposstr verbirgt, d.h. dieses Byte wird in den Akkumulator geladen.

Durch den nächsten Befehl STA \$FD wird dieses Byte dann in die Speicheradresse \$FD geschrieben.

Wie Sie sich sicher schon denken können, steht die Zeichenfolge #> dann für das höherwertige Byte der Adresse, welche durch das darauf folgende Label bezeichnet wird.

```
lda #>bitposstr
```

Durch den nächsten Befehl STA \$FE wird dieses Byte dann in die Speicheradresse \$FE geschrieben.

Soweit so gut, nun haben wir also die Adresse des Strings, welcher durch das Label bitposstr eingeleitet wird, in den Speicherstellen \$FD und \$FE. Sie steht also genau dort, wo sie das Unterprogramm printstr erwartet, wie sie im Kommentarblock nachlesen können.

Bevor wir das Unterprogramm printstr aufrufen können, müssen wir noch die gewünschten Werte für die Zeile und Spalte in die erforderlichen Register X und Y laden.

In unserem Fall hier soll der Inhalt des Strings bitposstr in Zeile 0 an der Spalte 9 erscheinen.

Darauf folgt exakt der gleiche Assemblercode, nur das diesmal auf den String, welcher durch das Label helpinfo eingeleitet wird, zugegriffen wird. Dieser wird in Zeile 22 an Spalte 8 ausgegeben.

Nachdem die beiden Strings ausgegeben wurden, musste ich den Cursor in die linke obere Ecke des Bildschirms versetzen, da durch die Ausgabe der READY-Meldung der Bildschirminhalt noch oben gescrollt wurde.

Durch die Versetzung des Cursors wird die READY-Meldung weiter oben ausgegeben und das Scrollen dadurch verhindert.

Nun können wir uns dem interessanten Teil des Unterprogramms printstr widmen:

```

; string ausgeben
ldy #$00
printstr_loop
lda ($fd),y
beq printstr_end
jsr $ffd2
iny
jmp printstr_loop
printstr_end
rts

```

Hier wird mit dem Befehl LDY #\$00 der Index 0 in das Y Register geladen, denn wir wollen bei der Stringausgabe ja bei Index 0, also mit dem ersten Zeichen, beginnen.

Nun folgt eine Schleife, in der mittels der Kernal-Funktion CHROUT, deren Aufruf durch den Befehl JSR \$FFD2 erfolgt, der String Zeichen für Zeichen ausgegeben wird. Die Funktion CHROUT erwartet den PETSCII-Code des auszugebenden Zeichens im Akkumulator und gibt das entsprechende Zeichen dann an der aktuellen Position des C64 Cursors aus.

Bei der Ausführung des Befehls LDA (\$FD),Y passiert nun folgendes:

Es wird die Speicheradresse, deren niederwertiges Byte in der Speicherstelle \$FD und deren höherwertiges Byte in der direkt darauf folgenden Speicherstelle \$FE hinterlegt ist, gebildet und der Index aus dem Y Register hinzuaddiert.

Die Adresse ergibt sich dann wie bereits erwähnt durch die Formel

Inhalt der Speicherstelle \$FD + 256 * Inhalt der Speicherstelle \$FE + Inhalt des Y Registers

Der gelb markierte Teil bleibt beim Durchlaufen der Schleife immer konstant. Es ändert sich nur der Inhalt des Y Registers, der bei jedem Schleifendurchlauf um 1 erhöht wird.

Beim ersten Schleifendurchlauf steht nach Ausführung des Befehls LDA (\$FD),Y der PETSCII-Code des Zeichens „7“ im Akkumulator, da dies das erste Zeichen im String bitposstr ist.

Durch die Anweisung BEQ printstr_end wird geprüft, ob der Inhalt des Akkumulators gleich 0 ist, also das Ende des Strings erreicht ist. Wenn dies der Fall ist, wird zum Label printstr_end gesprungen und durch den Befehl RTS findet der Rücksprung zum Aufrufer des Unterprogramms statt.

Ist das Ende des Strings jedoch noch nicht erreicht, dann wird über die Kernal-Funktion \$FFD2 das Zeichen ausgegeben. Im Anschluss wird der Index im Y Register um 1 erhöht und dann zum Label printstr_loop gesprungen.

Nun ergibt sich für den nächsten Schleifendurchlauf die Speicheradresse

Inhalt der Speicherstelle \$FD + 256 * Inhalt der Speicherstelle \$FE + Inhalt des Y Registers (nun 1)

Dadurch wird nun der PETSCII-Code des zweiten Zeichens geladen, beim nächsten Schleifendurchlauf dann jener des dritten Zeichens usw. bis das Ende des Strings erreicht ist.

Indem wir dem Unterprogramm printstr also in den Speicherstellen \$FD / \$FE die Speicheradresse bekannt geben, an der es den auszugebenden String findet, können wir das Unterprogramm universal einsetzen und jeden beliebigen String ausgeben, weil wir innerhalb des Unterprogramms nicht an Namen wie bitposstr oder helpinfo gebunden sind.

Wenn Sie das Programm starten, sollte sich folgendes Bild zeigen:



Als Nächstes möchte ich Ihnen erklären, wie ich die eigentliche Editorfläche, also die Nummerierung der Reihen gefolgt von den kleinen Punkten ausgegeben habe.

Dazu habe ich das Programm PRINTROWS erstellt.

Da sich die 21 Strings nur durch die Nummer der Reihe am Anfang des Strings unterscheiden, habe ich zunächst einen String namens editorrowstr definiert, welcher 26 Zeichen umfasst und im Datenbereich am Ende des Programms zu finden ist.

Die ersten beiden Zeichen in diesem String sind als Platzhalter für die Nummer der Reihe gedacht.

In einer Schleife durchlaufe ich dann die Reihenummer von 0 bis 20, platziere diese in den ersten beiden Stellen des Strings und gebe diesen String dann aus.

Die Reihennummern selbst habe ich ebenfalls in Form von Strings abgelegt.
Nachfolgend die Definitionen der soeben erwähnten Strings.

```
editorrowstr
.text " "
.text " "
.text " "
.null " "
rownrstr
.text "00"
.text "01"
.text "02"
.text "03"
.text "04"
.text "05"
.text "06"
.text "07"
.text "08"
.text "09"
.text "10"
.text "11"
.text "12"
.text "13"
.text "14"
.text "15"
.text "16"
.text "17"
.text "18"
.text "19"
.text "20"
```

Am Anfang des Strings namens editorrowstr sind die beiden Platzhalter-Positionen und darauffolgend die 24 Punkte (3 Bytes entsprechen 24 Bits, daher 24 Punkte) zu sehen.

Die letzte Gruppe von Punkten habe ich mit .null abgeschlossen, damit automatisch ein Nullbyte angehängt wird. Bei den beiden anderen Gruppen war dies nicht nötig (und auch nicht vorgesehen), da alle 24 Punkte ja nebeneinander in einer Reihe liegen.

Im Anschluss sind die Strings zu sehen, welche die Reihennummern enthalten. Bei diesen ist kein Nullbyte am Ende notwendig, da diese Strings ja nicht direkt am Bildschirm ausgegeben werden, sondern nur ausgelesen und dann in die beiden Platzhalter-Stellen im String editorrowstr kopiert werden.

Bevor ich Ihnen nun den Assemblercode zur Ausgabe der Editor-Reihen erkläre, möchte ich noch ein Detail zum Sprite-Editor erwähnen.

Ich habe den Editor nicht an einer fixen Position am Bildschirm platziert, sondern ich habe mir die Möglichkeit offen gehalten, dessen Position am Bildschirm frei zu wählen, soweit das innerhalb der begrenzten Bildschirm-Abmessungen möglich ist.

Möglich wird dies durch die Definition zweier Variablen namens editorrow und editorcol.

Sie enthalten die Position der linken oberen Ecke des Sprite-Editors.

```
editorrow
.byte $00
editorcol
.byte $07
```

Die Strings aus denen der Sprite-Editor besteht, werden dann nicht an einer fixen Position am Bildschirm ausgegeben, sondern deren Positionen werden anhand der Werte von editorrow und editorcol relativ zu dieser Position berechnet.

Nachfolgend sehen Sie durch das grüne Kästchen markiert, welche Position gemeint ist.



Im Sprite-Editor habe ich für editorrow den Wert 0 und für editorcol den Wert 7 eingestellt, damit er mittig am Bildschirm dargestellt wird.

Der String, welcher die Bitpositionen am oberen Rand darstellt, wird beispielsweise zwei Stellen rechts von dieser Position ausgegeben.

Die Reihen beginnen genau eine Zeile unterhalb dieser Position.

Der Hinweis auf die Hilfeinformation wird an der Position editorrow + 23 und editorcol + 1 ausgegeben.

Würde ich nun den Wert von editorrow um 1 erhöhen, dann würden alle Elemente des Editors ebenfalls um eine Zeile nach unten wandern.

Dasselbe gilt für editorcol, würde ich den Wert auf 0 ändern, dann würde der gesamte Editor nun am linken Bildschirmrand angezeigt werden.

Dies hat den Vorteil, dass man sich keine Gedanken um die Verschiebung der einzelnen Strings machen muss, da deren Positionen ausgehend von der linken oberen Ecke des Editors berechnet werden.

Warum habe ich diesen Ansatz gewählt? Nun ja, es könnte ja sein, dass man nachträglich noch zusätzliche Elemente am Bildschirm platzieren will und dafür den Editor entsprechend verschieben müsste. Hätte ich eine fixe Position für den Editor festgelegt, dann wäre eine nachträgliche Verschiebung recht aufwändig, da ich ja alle Elemente separat verschieben müsste.

Durch den Ansatz mit der frei wählbaren Position kann ich z.B. durch die Angabe von `editorrow = 0` und `editorcol = 0` den gesamten Editor sofort auf die linke obere Ecke des Bildschirms verlagern.

Kommen wir nun zum Assemblercode, mit dem ich die einzelnen Editorreihen am Bildschirm ausgabe.

Folgende Variable spielt bei der Ausgabe der Reihen auch noch eine wichtige Rolle:

```
scrow      .byte $00
```

Sie enthält beim Durchlaufen der Schleife immer jene Zeile, in der die aktuelle Reihe ausgegeben wird.

Wie Sie nachfolgend gleich zu Beginn des Codes sehen können, wird der Inhalt der Variablen `editorrow` in den Akkumulator geladen, dessen Inhalt um 1 erhöht und das Ergebnis in der Variablen `scrow` gespeichert, d.h. die erste Reihe wird in der Zeile `editorrow + 1` ausgegeben.

```
;-----  
; hauptprogramm  
; hier wird das programm gestartet  
; start von basic aus mit sys 12288  
  
    *= $3000  
  
    ; editorreihen ausgeben  
    lda editorrow  
    clc  
    adc #$01  
    sta scrow  
  
    lda #$00  
printrowloop  
    ; aktuelle reihennr (0..20)  
    ; auf dem stack merken  
    pha  
  
    ; reihennr * 2 = index fuer  
    ; feld rownrstr  
    asl a  
    tax  
  
    ; erste ziffer an der ersten  
    ; stelle im string  
    ; editorrowstr eintragen  
    lda rownrstr,x  
    sta editorrowstr  
  
    ; zweite ziffer an der zweiten
```

```

; stelle im string
; editorrowstr eintragen
inx
lda rownrstr,x
sta editorrowstr+1
; fertigen string ausgeben
lda #<editorrowstr
sta $fd
lda #>editorrowstr
sta $fe
ldx scrow
ldy editorcol
jsr printstr
; reihennr wieder vom
; stack holen
pla
; ist bereits die letzte
; reihe erreicht?
cmp #$14
; wenn ja, schleife beenden
beq printrowloopend
; wenn nicht => weiter mit
; naechster reihe
clc
adc #$01
inc scrow
jmp printrowloop
printrowloopend
rts

```

Beginnen wir ab dem Label printrowloop mit der Erklärung. Ich habe den Akkumulator hier verwendet, um die aktuelle Reihennummer zu speichern und deswegen habe ich ihn vor Eintritt in die Ausgabeschleife durch den Befehl LDA #\$00 mit der Nummer der ersten Reihe geladen.

Der Akkumulator wird während der Schleife auch noch für andere Zwecke verwendet und deswegen muss ich den aktuellen Inhalt, also die aktuelle Reihennummer, auf dem Stack sichern.

Im nächsten Schritt wird der Inhalt des Akkumulators nämlich bereits durch den Befehl ASL mit zwei multipliziert. Warum? Durch die Multiplikation der Reihennummer mit 2 ergibt sich ein Index in das Datenfeld rownrstr, an dem der Nummernstring für die jeweilige Reihe steht.

Also z.B. 00 für die Reihe 0, 01 für die Reihe 1 usw.

Die erste Ziffer dieses Nummernstrings wird dann an die erste Position im String editorrowstr eingesetzt und die zweite Ziffer wird analog dazu an die zweite Position eingesetzt.

In folgender Tabelle ist dargestellt, wie ausgehend von der Reihennummer durch die Multiplikation mit zwei der Index in das Datenfeld rownrstr berechnet wird und der ausgelesene Nummernstring dann letztendlich in den String editorrowstr eingesetzt wird.

Reihennummer	Index in Datenfeld rownrstr	Nummernstring aus Datenfeld rownrstr	editorrowstr
0	0	00	00.....
1	2	01	01.....
2	4	02	02.....
3	6	03	03.....
4	8	04	04.....
5	10	05	05.....
6	12	06	06.....
7	14	07	07.....
8	16	08	08.....
9	18	09	09.....
10	20	10	10.....
11	22	11	11.....
12	24	12	12.....
13	26	13	13.....
14	28	14	14.....
15	30	15	15.....
16	32	16	16.....
17	34	17	17.....
18	36	18	18.....
19	38	19	19.....
20	40	20	20.....

Den errechneten Index kopieren wir mit dem Befehl TAX in das X Register, damit wir über die X indizierte Adressierung auf den Nummernstring im Datenfeld rownrstr zugreifen können.

Durch den Befehl LDA rownrstr,x laden wir das erste Zeichen des Nummernstrings und kopieren es mit dem Befehl STA editorrowstr an die erste Stelle im String editorrowstr.

Nun erhöhen wir durch den Befehl INX den Index im X Register um 1, sodass wir auf das zweite Zeichen im Nummernstring zugreifen können.

Durch den Befehl LDA rownrstr,x laden wir dieses in den Akkumulator und kopieren es mit dem Befehl STA editorrowstr+1 an die zweite Stelle im String editorrowstr.

Nun ist der Ausgabestring für die Reihe fertig und wir können ihn mit dem Unterprogramm printstr ausgeben.

Dazu laden wir zuerst das niederwertige Byte der Adresse des Strings editorrowstr in den Akkumulator und kopieren es von dort in die Speicherstelle \$FD.

Dasselbe machen wir mit dem höherwertigen Byte der Adresse und kopieren es in die Speicherstelle \$FE.

Nun müssen wir noch angeben, wo der String ausgegeben werden soll. Die Zeile, in der die aktuelle Reihe ausgegeben wird, steht wie eingangs erwähnt in der Variablen scrow.

Daher kopieren wir deren Inhalt mit dem Befehl LDX scrow in das X Register.

Die Spalte für die Ausgabe der Reihen ist für jede Reihe identisch und entspricht dem Wert der Variablen editorcol.

Durch den Befehl LDY editorcol laden wir diesen Wert in das Y Register und können nun die aktuelle Reihe durch Aufruf des Unterprogramms printstr ausgeben.

Nun brauchen wir wieder die Nummer der aktuellen Reihe. Zu Beginn des Unterprogramms haben wir diese auf dem Stack gesichert und holen sie uns nun mit dem Befehl PLA wieder vom Stack.

Durch den Befehl CMP #\$14 prüfen wir, ob wir bereits die letzte Reihe, also jene mit der Nummer 20 erreicht haben. Wenn ja, dann sind alle Reihen ausgegeben und wir können mit dem Befehl BEQ printrowloopend die Schleife verlassen und zum Label printrowloopend springen.

Dort wird das Programm dann durch den Befehl RTS beendet und zu Basic zurückgekehrt.

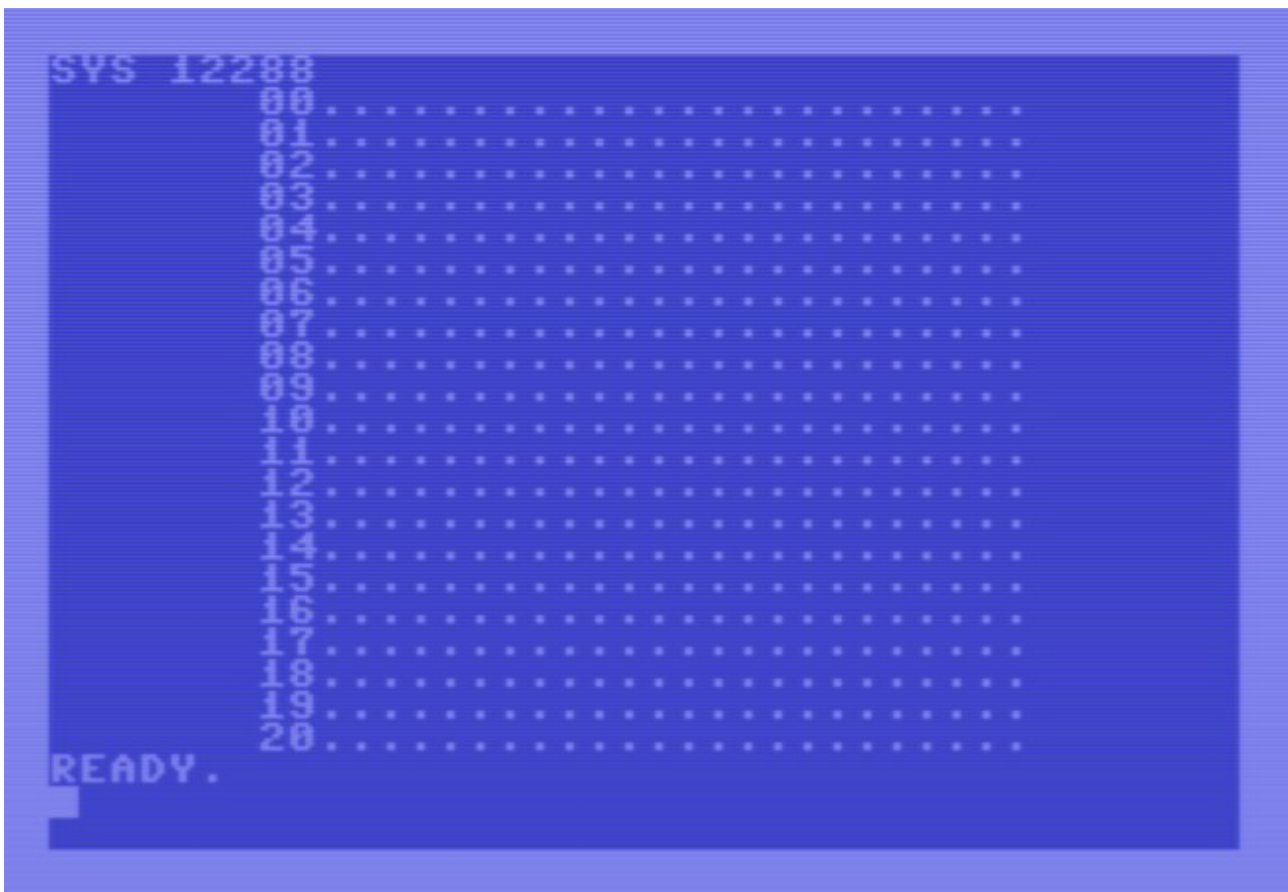
Falls jedoch noch nicht alle Reihen ausgegeben wurden, erhöhen wir die Reihenummer im Akkumulator um 1, um zur nächsten Reihe überzugehen.

Dasselbe machen wir mit dem Inhalt der Variablen scrow, denn die nächste Reihe steht ja eine Zeile unter der aktuell ausgegebenen Reihe.

Abschließend springen wir wieder zurück zum Anfang der Schleife.

Dieser Ablauf wiederholt sich für alle Reihen, sodass eine Reihe nach der anderen am Bildschirm ausgegeben wird.

Wenn das Programm durchgelaufen ist, sollte sich folgendes Bild zeigen:



Im nächsten Programm DRAWEDITOR zeige ich Ihnen, wie ich die Ausgabe des Sprite-Editors vervollständigt und zwecks der besseren Übersicht in ein eigenes Unterprogramm namens draweditor verlagert habe.

Im Hauptteil wird eigentlich nur dieses Unterprogramm aufgerufen, welches die komplette Benutzeroberfläche des Sprite-Editors ausgibt. Diesesmal sind nicht nur die nummerierten Editorreihen dabei, sondern auch der String am oberen Rand, welcher die Bitpositionen darstellt und auch der String, der den Hinweis auf die verfügbaren Hilfsinformationen enthält.

Nachdem die Benutzeroberfläche des Sprite-Editors ausgegeben wurde, musste ich wie bereits beim vorherigen Programm den Cursor in die linke obere Ecke des Bildschirms versetzen, da durch die Ausgabe der READY-Meldung der Bildschirminhalt noch oben gescrollt wurde.

Durch die Versetzung des Cursors wird die Meldung weiter oben ausgegeben und das Scrollen dadurch verhindert.

```

:-----
: hauptprogramm
: hier wird das programm gestartet
: start von basic aus mit sys 12288
:
:      *= $3000
:
:      ; benutzeroberflaeche anzeigen
:      jsr draweditor
:
:      ; cursor in die linke obere
:      ; ecke versetzen, damit die
:      ; ausgabe nicht nach oben
:      ; gescrollt wird

```

```

        clc
        ldx #$00
        ldy #$00
        jsr $fff0

        rts

;-----
; draweditor
; zeichnet das userinterface
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
draweditor
        ; bildschirm loeschen

        lda #$93
        jsr $ffd2

        ; string mit bitpositionen
        ; ausgeben

        lda #<bitposstr
        sta $fd

        lda #>bitposstr
        sta $fe

        ldx editorrow
        ldy editorcol
        iny
        iny

        jsr printstr

        ; editorreihen ausgeben

        lda editorrow
        clc
        adc #$01
        sta scrow

        lda #$00
printrowloop
        ; aktuelle reihennr (0..20)
        ; auf dem stack merken

        pha

        ; reihennr * 2 = index fuer
        ; feld rownrstr

        asl a

        tax

        ; erste ziffer an der ersten
        ; stelle im string
        ; editorrowstr eintragen

        lda rownrstr,x
        sta editorrowstr

        ; zweite ziffer an der zweiten
        ; stelle im string
        ; editorrowstr eintragen

```

```

        inx
        lda rownrstr,x
        sta editorrowstr+1
        ; fertigen string ausgeben
        lda #<editorrowstr
        sta $fd
        lda #>editorrowstr
        sta $fe
        ldx scrow
        ldy editorcol
        jsr printstr
        ; reihennr wieder vom
        ; stack holen
        pla
        ; ist bereits die letzte
        ; reihe erreicht?
        cmp #$14
        ; wenn ja, schleife beenden
        beq printrowloopend
        ; wenn nicht => weiter mit
        ; naechster reihe
        clc
        adc #$01
        inc scrow
        jmp printrowloop
printrowloopend
        ; hinweis auf verfuegbare
        ; hilfsinformationen anzeigen
        lda #<helpinfo
        sta $fd
        lda #>helpinfo
        sta $fe
        lda editorrow
        clc
        adc #$17
        tax
        ldy editorcol
        iny
        jsr printstr
        rts

```

Wenn Sie das Programm starten, sollte sich folgendes Bild zeigen:



Gratulation! Die Ausgabe der Benutzeroberfläche haben wir schon mal geschafft!

Experimentieren Sie ruhig ein wenig mit den Variablen `editorrow` und `editorcol`.

Es bleibt aufgrund der Größe des Sprite-Editors zwar nicht viel Spielraum für Veränderungen, aber vielleicht finden Sie ja eine Position, die Ihnen besser gefällt als die von mir gewählte.

Als Nächstes wollen wir uns Schritt für Schritt den Programmfunktionen widmen, welche über das Drücken einer bestimmten Taste oder einer Tastenkombination ausgelöst werden.

Doch bevor wir damit beginnen, müssen wir noch einige Aufgaben erledigen, welche mit der variablen Positionierung des Editors zu tun haben.

Ich habe vorhin erwähnt, dass sämtliche Positionsangaben auf Basis der beiden Variablen `editorrow` und `editorcol` berechnet werden. Dies betrifft auch die Cursor-Steuerung, denn der Cursor soll sich ja sinnvollerweise nur innerhalb des Editorbereichs bewegen lassen.

Es muss in Bezug auf den Cursor daher für den Zeilen- und Spaltenwert ein Minimal- und Maximalwert festgelegt werden.

Bei jeder Cursorbewegung muss daher geprüft werden, ob durch diese die Grenzen des Editors eingehalten werden. Diese Grenzen sind jedoch abhängig von den Inhalten der Variablen `editorrow`

und editorcol und deswegen werden wir ein Unterprogramm namens init schreiben, das beim Starten des Editors genau einmal aufgerufen wird und sämtliche Positionsangaben mit den korrekten Werten initialisiert.

Im Zusammenhang mit der Cursorsteuerung werden wir daher einige neue Variablen benötigen, welche die Grenzen definieren, innerhalb derer sich der Cursor bewegen darf.

Name	Inhalt	Berechnung
csrrow	Zeile in der sich der Cursor aktuell befindet	
csrcol	Spalte in der sich der Cursor aktuell befindet	
csrminrow	Minimalwert für den Zeilenwert des Cursors, darf durch eine Bewegung nach oben nicht unterschritten werden	editorrow + 1
csrmaxrow	Maximalwert für den Zeilenwert des Cursors, darf durch eine Bewegung nach unten nicht überschritten werden	editorrow + 20
csrmincol	Minimalwert für den Spaltenwert des Cursors, darf durch eine Bewegung nach links nicht unterschritten werden	editorcol + 2
csrmaxcol	Maximalwert für den Spaltenwert des Cursors, darf bei einer Bewegung nach rechts nicht überschritten werden	editorcol + 23

Der Wert von csrminrow entspricht also der Zeile, in der sich die Reihe 00 befindet und der Wert von csrmaxrow entspricht der Zeile, in der sich die Reihe 20 befindet.

Der Wert von csrmincol entspricht jener Spalte, die rechts neben der Reihenummer liegt und der Wert von csrmaxcol entspricht jener Spalte, welche dem rechten Rand des Editors entspricht. Parallel zur aktuellen Position des Cursors in Form von Zeilen- und Spaltenwert, wird in den Speicherstellen \$FB und \$FC dessen aktuelle Adresse im Bildschirmspeicher mitgeführt.

Um den Cursor an seiner aktuellen Position darzustellen, muss das Zeichen ja an die richtige Adresse im Bildschirmspeicher geschrieben werden.

Es würde sehr viel Zeit kosten, diese Adresse bei jeder Cursorbewegung mittels des Unterprogramms calcposaddr neu zu berechnen und deswegen wird sie nach jeder Cursorbewegung über sehr viel einfachere Rechenoperationen aktualisiert.

Der aktuelle Wert steht dann jederzeit in den Speicherstellen \$FB und \$FC zur Verfügung, wobei in der Speicherstelle \$FB das niederwertige und in der Speicherstelle \$FC das höherwertige Byte dieser Adresse steht.

Ich werde dies alles später noch im Detail erklären, wenn wir konkret zur Umsetzung der Cursorsteuerung kommen.

Im Zuge der Ausführung einiger Programmfunktionen, zu denen wir erst später kommen werden, ist es sinnvoll, den Cursor wieder auf die Ausgangsposition zu versetzen (also in die linke obere Ecke des Editorbereichs) und um diese Adresse nicht immer wieder neu berechnen zu müssen, merken wir uns diese in der Variablen csrhomeaddr.

Soweit so gut, die Liste mit den Variablen, die innerhalb des Unterprogramms angesprochen werden, ist nun komplett und wir können mit der Umsetzung dieses Unterprogramms beginnen.

Nachfolgend der Assembler-Code des Unterprogramms init:

```
-----
: init
: berechnet die grenzen fuer die
: cursorbewegung sowie die startadresse
: des cursors im bildschirmspeicher,
: die startposition des cursors wird
: ebenfalls gesetzt
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,y,status
init
; grenzen fuer zeilenwert
; des cursors berechnen und
; zeilenwert fuer dessen
; startposition setzen

lda editorrow
clc
adc #$01
sta csrminrow
sta csrrow
adc #$14
sta csrmaxrow

; grenzen fuer spaltenwert
; des cursors berechnen und
; spaltenwert fuer dessen
; startposition setzen

lda editorcol
adc #$02
sta csrmincol
sta csrcol
adc #$17
sta csrmaxcol

; adresse des cursors im
; bildschirmspeicher
; berechnen

ldx csrcol
ldy csrrow
jsr calcposaddr

; diese wird laufend bei
```

```

; jeder cursorbewegung
; in den speicherstellen
; $fb und $fc aktualisiert

stx $fb
sty $fc

; startposition des cursors
; merken

stx csrhomeaddr
sty csrhomeaddr+1

rts

```

Hier wird zunächst der Inhalt der Variablen editorrow in den Akkumulator geladen, dessen Inhalt um 1 erhöht und das Ergebnis in die Variablen csrminrow und csrrow geschrieben. In die Variable csrrow deswegen, weil sich der Cursor zu Beginn des Programms ja ebenfalls in dieser Zeile befindet.

Im Anschluss wird der Inhalt der Variablen csrmaxrow berechnet, indem zum Inhalt des Akkumulator der Wert 20 addiert wird.

Analog dazu erfolgen nun gemäß der obigen Tabelle dieselben Berechnungen für die Variablen csrmincol, csrcol und csrmaxcol.

Da nun in den Variablen csrrow und csrcol die Startposition des Cursors steht, können wir nun durch das Unterprogramm calcposaddr die Adresse im Bildschirmspeicher berechnen, welche der Startposition des Cursors entspricht. Das niederwertige Byte steht laut Dokumentation, welche wir beim Unterprogramm calcposaddr hinterlegt haben, in der Speicherstelle \$FB und das höherwertige Byte in der Speicherstelle \$FC.

Nachdem die Adresse berechnet ist, merken wir sie uns wie bereits erwähnt in der Variablen csrhomeaddr. Das niederwertige Byte steht dann in der Speicherstelle csrhomeaddr und das höherwertige Byte in der Speicherstelle csrhomeaddr + 1.

Und das war's auch schon mit dem Unterprogramm init, d.h. wir können uns voller Elan der Umsetzung der Programmfunktionen widmen. Wir beginnen mit der Cursorsteuerung und der Beendigung des Sprite-Editors.

Doch bevor wir die Tastatur abfragen können, müssen wir zunächst die Tastaturcodes der gewünschten Tasten(kombinationen) ermitteln. Diese Codes können wie beispielsweise die Farbcodes in Tabellen nachgeschlagen werden. Ich habe diejenigen, welche für den Sprite-Editor relevant sind, herausgesucht und in folgender Tabelle zusammengefasst:

Funktion	Tasten(kombination)	Tastaturcode
Cursor nach rechts bewegen	Cursor nach rechts	\$1D
Cursor nach unten bewegen	Cursor nach unten	\$11
Cursor nach links bewegen	Cursor nach links	\$9D
Cursor nach oben bewegen	Cursor nach oben	\$91
Bit setzen	Return Taste	\$0D
Bit löschen	Leertaste	\$20
Sprite in Datei speichern	SHIFT + Taste S	\$D3
Sprite aus Datei laden	SHIFT + Taste L	\$CC
Editorbereich löschen	SHIFT + CLR/HOME	\$93

Hilfsinformationen anzeigen	SHIFT + Taste H	\$C8
Editor beenden	SHIFT + Taste Q	\$D1

Für diese Tastendefinitionen legen wir im Datenabschnitt am Ende des Programms eigene Variablen an.

```

csrrightkey    .byte $1d
csrdownkey     .byte $11
csrleftkey     .byte $9d
csrupkey       .byte $91
setbit1key     .byte $0d
setbit0key     .byte $20
savekey        .byte $d3
loadkey        .byte $cc
clearkey       .byte $93
helpkey        .byte $c8
quitkey        .byte $d1

```

Durch diese Namen wird der Assembler-Code lesbarer, da wir anstelle der Tastaturcodes sprechende Namen verwenden.

Ein Vergleich wie `CMP csrrightkey` liest sich einfacher als `CMP #$1D` oder?

Wir haben durch die Verwendung dieser Namen auch die Möglichkeit, die Zuordnung der Tastenkombinationen zu den Programmfunktionen jederzeit zu ändern.

Wenn wir den Editor beispielsweise nicht mehr über die Tastenkombination SHIFT + Taste Q, sondern durch die Taste X beenden wollen, dann müssten wir beispielsweise nur folgende Änderung in obigem Code durchführen:

```

quitkey        .byte $58

```

Wir werden die Tastaturabfrage in ein eigenes Unterprogramm namens `keyctrl` auslagern. Dieses wird beim Start des Programms aufgerufen und erst wieder verlassen, wenn der Benutzer die Tastenkombination zur Beendigung des Editors drückt.

Beginnen wir ganz einfach und setzen zunächst mal genau diese Funktion, also die Beendigung des Editors, um.

```

:-----:
: keyctrl
: fragt die tastatur ab und verzweigt
: in die einzelnen programmfunktionen
: wenn die entsprechende taste
: gedrueckt wird
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,status
:
keyctrl
keyloop
    ; tastatur abfragen bis eine
    ; taste gedrueckt wird
    jsr $ffe4
    beq keyloop
    ; editor beenden?
    cmp quitkey
    beq keyctrl_end
    ; ansonsten wieder mit der
    ; tastaturabfrage fortsetzen
    jmp keyloop
keyctrl_end
    ; vor dem beenden des editors
    ; noch den bildschirm loeschen
    ; dies geschieht durch ausgabe
    ; des zeichens mit dem
    ; code 147 (clear)
    lda scrcode_clrscr
    jsr $ffd2
    rts

```

Zur Tastaturabfrage verwende ich hier die Kernal-Funktion GETIN, welche über die Adresse \$FFE4 aufgerufen werden kann.

Diese Funktion prüft, ob eine Taste gedrückt wurde. Wenn ja, liefert sie den Tastencode der gedrückten Taste im Akkumulator zurück. Falls keine Taste gedrückt wurde, wird dies durch den Wert 0 im Akkumulator signalisiert.

Wir müssen also die Tastatur zunächst solange abfragen, bis eine Taste gedrückt wird. Dies wird durch die Anweisung BEQ keyloop nach dem Aufruf der Funktion GETIN erreicht, da im Falle des Inhalts 0 im Akkumulator zum Label keyloop gesprungen und somit die Tastatur erneut abgefragt wird.

Wenn wir eine beliebige Taste drücken, dann steht deren Tastaturcode im Akkumulator. Durch den Vergleich CMP quitkey wird geprüft, ob wir die Taste zum Beenden des Editors gedrückt haben.

Ist dies der Fall, dann ergibt der Vergleich im Akkumulator den Wert 0 und wir können durch die Anweisung BEQ keyctrl_end zum Label keyctrl_end springen.

Dort wird unter Verwendung der Kernal-Funktion CHROUT (JSR \$FFD2) das Zeichen mit dem Code 147 (\$93) ausgegeben, was bekanntlich ein Löschen des Bildschirms bewirkt.

In Basic würde man PRINT CHR\$(147) schreiben.

Auch hier habe ich anstelle des Codes einen Namen verwendet, welchen ich im Datenabschnitt angegeben habe:

```
scrcode_clrscr  
    .byte $93
```

Danach wird durch den Befehl RTS zum Aufrufer zurückgekehrt.

Wie Sie nachfolgend sehen können, habe ich den Hauptteil des Programms um den Aufruf des Unterprogramms keyctrl erweitert:

```
-----  
; hauptprogramm  
; hier wird das programm gestartet  
; start von basic aus mit sys 12288  
  
    *= $3000  
  
    ; editor variablen  
    ; initialisieren  
  
    jsr init  
    ; benutzeroberflaeche anzeigen  
  
    jsr draweditor  
  
    ; tastaturabfrage starten  
  
    jsr keyctrl  
  
    rts
```

Auf den Aufruf JSR keyctrl folgt nur mehr der Befehl RTS, d.h. der Editor wird beendet und zu Basic zurückgekehrt.

Falls wir nicht die Taste für die Beendigung des Editors gedrückt haben, wird durch den Befehl JMP keyloop wieder zum Label keyloop gesprungen und die Tastatur erneut abgefragt, bis eine Taste gedrückt wird.

Ich habe den aktuellen Stand des Programms unter dem Namen QUIT bereitgestellt. Laden Sie das Programm im TMP und probieren Sie es aus. Nach dem Start wird die Benutzeroberfläche des Editors angezeigt und auf einen Tastendruck gewartet.

Wenn Sie die Tastenkombination SHIFT + Q drücken, wird wie vorhin beschrieben, der Bildschirm gelöscht und der Editor beendet. Wenn Sie andere Tasten drücken, erfolgt aktuell noch keine Reaktion.

Nachdem der Editor beendet wurde, können Sie ihn jederzeit wieder mit SYS 12288 starten.

Bevor wir nun zur Umsetzung der Cursorsteuerung kommen, müssen wir uns noch mit dem Gegenstück zur 16bit Addition, der 16bit Subtraktion beschäftigen.

Subtraktion zweier 16bit Zahlen

Wiederholen wir zunächst die Subtraktion zweier 8bit Zahlen, welche durch den Befehl SBC durchgeführt wird. Der Befehl SBC #\$10 subtrahiert beispielsweise den Wert \$10 vom Inhalt des Akkumulators. Zusätzlich wird noch der umgekehrte Inhalt des Carry Flags subtrahiert.

Daher ist es wichtig, vor der Ausführung des Befehls SBC das Carryflag mit dem Befehl SEC zu setzen. Durch die Umkehrung des Inhalts fließt es dann mit dem Wert 0 in die Subtraktion mit ein, hat also keine Auswirkung auf das Ergebnis.

Bei der Addition war es umgekehrt, hier musste vor der Ausführung des Befehls ADC das Carryflag mit dem Befehl CLC gelöscht werden.

Bei der Subtraktion kann es im Gegensatz zur Addition nicht zu einem Überlauf kommen. Stattdessen kann es jedoch zu einem Unterlauf kommen, wenn man also vom Inhalt des Akkumulators einen größeren Wert subtrahiert, als er selbst enthält.

Solch ein Unterlauf wird durch ein gelöschtes Carryflag signalisiert.

Nachfolgend sehen Sie den Assembler-Code für das Unterprogramm sbc16, welches die 16bit Version des Befehls SBC darstellt.

```
-----  
: sbc16  
: subtrahiert zwei 16bit zahlen  
: zahl1-zahl2  
:  
: parameter:  
: zahl1: lo/hi in $fb/$fc  
: zahl2: lo/hi im x/y register  
:  
: rueckgabewerte:  
: differenz: lo/hi in $fb/$fc  
:  
: aendert:  
: a,status,$fd,$fe  
:  
sbc16  
    stx $fd  
    sty $fe  
    ; 10 bytes subtrahieren  
    sec  
    lda $fb  
    sbc $fd  
    sta $fb  
    ; hi bytes subtrahieren  
    lda $fc  
    sbc $fe  
    sta $fc  
    rts
```

Hier wird zunächst das niederwertige Byte der zweiten Zahl vom X Register in die Speicherstelle \$FD kopiert. Dasselbe geschieht mit dem höherwertigen Byte der zweiten Zahl, es wird vom Y Register in die Speicherstelle \$FE kopiert.

Der Grund für das Umkopieren ist der, dass es leider nicht möglich ist, Inhalte von Registern direkt voneinander zu subtrahieren. Man kann also vom Inhalt des Akkumulators nicht auf direktem Wege den Inhalt des X Registers oder Y Registers subtrahieren, sondern muss den Umweg über eine Speicherstelle nehmen.

Dasselbe gilt auch für die Addition, auch hier ist es auf direktem Wege nicht möglich, den Inhalt des X Registers oder Y Registers zum Inhalt des Akkumulators zu addieren.

Analog zur 16bit Addition werden hier zunächst die niederwertigen Bytes der beiden Zahlen subtrahiert. Das Ergebnis wird in das niederwertige Byte der Differenz, also in die Speicherstelle \$FB, geschrieben.

Dann werden die beiden höherwertigen Bytes der beiden Zahlen subtrahiert und das Ergebnis in das höherwertige Byte der Differenz, also in die Speicherstelle \$FC, geschrieben.

Bei der Subtraktion der niederwertigen Bytes kann es zu einem Unterlauf kommen, was durch ein gelöscht Carryflag angezeigt werden würde.

Falls es zu einem Unterlauf kam, das Carryflag also gelöscht wurde, dann fließt dieses durch die Umkehrung des Inhalts mit dem Wert 1 in die Subtraktion der höherwertigen Bytes mit ein.

Darstellung und Steuerung des Cursors

Was den Cursor selbst betrifft, habe ich mich dafür entschieden, dem Schema des nativen Cursors zu folgen. Der Cursor wird durch kein eigenes Zeichen dargestellt, sondern er wird dadurch sichtbar gemacht, dass das Zeichen „unter“ dem Cursor revers dargestellt wird.

Bewegt man den Cursor weiter, wird das Zeichen wieder normal dargestellt und das Zeichen auf der neuen Cursorposition wird nun revers dargestellt. Dieser Wechsel von revers zu nicht-revers findet bei jeder Cursorbewegung statt.

Wenn Sie den Editor bereits ausprobiert haben, können Sie dies gut beobachten.

Glücklicherweise lässt sich die Umschaltung zwischen reverser und nicht-reverser Darstellung eines Zeichens relativ einfach umsetzen, da man den Screencode des einen ganz einfach aus dem Screencode des anderen errechnen kann.

Man muss nur den Wert 128 addieren (oder umgekehrt subtrahieren), je nachdem zwischen welchen Darstellungen man umschalten will.

Die Addition ist bei der Umschaltung von der nicht-reversen zur reversen Darstellung nötig und umgekehrt die Subtraktion bei der Umschaltung von der reversen zur nicht-reversen Darstellung. Das Zeichen „A“ hat beispielsweise den Screencode 1 und das reverse Zeichen „A“ hat den Screencode 129.

$1 + 128$ ergibt 129 und umgekehrt ergibt $129 - 128$ wieder 1

Mit diesem Wissen ausgestattet, können wir uns nun zwei Unterprogramme schreiben.

Das erste Unterprogramm namens showcsr soll den Cursor an seiner aktuellen Position darstellen, also das Zeichen an dieser Position revers darstellen.

Das zweite Unterprogramm namens removecsr soll den Cursor an seiner aktuellen Position entfernen, d.h. das Zeichen an dieser Position wieder in nicht-reverser Darstellung anzeigen.

Diese Umsetzungen sind überhaupt nicht schwierig, denn die Adresse im Bildschirmspeicher, an der der Cursor aktuell steht, ist in den beiden Speicherstellen \$FB und \$FC vermerkt.

Wir müssen also nur den Wert auslesen, welcher an dieser Speicheradresse zu finden ist, je nach dem entweder 128 addieren oder subtrahieren und das Ergebnis wieder zurück an diese Speicheradresse schreiben.

Nachfolgend der Assembler-Code des Unterprogramms showcsr:

```
-----  
: showcsr  
: zeigt den cursor an jener adresse  
: im bildschirmspeicher an welche in  
: den speicherstellen $fb und $fc  
: hinterlegt ist. das zeichen an dieser  
: position wird revers dargestellt.  
:  
: parameter:  
: keine  
:  
: rueckgabewerte:  
: keine  
:  
: aendert:  
: a,y,status  
:  
showcsr  
    ldy #$00  
    lda ($fb),y  
    clc  
    adc #$80  
    sta ($fb),y  
    rts
```

Über die neue Adressierungsart, welche wir vor kurzem kennengelernt haben, wird hier der Inhalt aus jener Speicheradresse gelesen, an der der Cursor gerade steht. Diese Adresse ist, wie vorhin erwähnt, in den Speicherstellen \$FB bzw. \$FC zu finden und durch den Befehl LDA (\$FB),Y wird der dortige Inhalt in den Akkumulator geladen.

Nun wird durch den Befehl ADC #\$80 der Wert 128 zum Inhalt des Akkumulators addiert und im nächsten Befehl STA (\$FB),Y wieder zurück an die aktuelle Adresse des Cursors geschrieben.

Dadurch wird das Zeichen, das sich aktuell dort befindet, revers dargestellt.

Das Unterprogramm removecsr funktioniert absolut identisch, nur mit dem Unterschied, das hier der Wert 128 nicht addiert, sondern subtrahiert wird. Dadurch wird das Zeichen, welches sich aktuell an der Adresse des Cursors befindet, wieder in nicht-reverser Darstellung angezeigt.

```
-----  
: removecsr  
: loescht den cursor an jener adresse  
: im bildschirmspeicher welche in den  
: speicherstellen $fb und $fc
```

```

; hinterlegt ist. das zeichen an dieser
; position wird wieder nicht-revers
; dargestellt.
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,y,status
removecsr
    ldy #$00
    lda ($fb),y
    sec
    sbc #$80
    sta ($fb),y
    rts

```

Das Unterprogramm showcsr werden wir auch gleich aufrufen, denn wenn der Editor gestartet wird, soll der Cursor an seiner Ausgangsposition angezeigt werden.

Ich habe daher das Unterprogramm draweditor um einen Aufruf von showcsr ergänzt. Diesen habe ich direkt vor dem Befehl RTS platziert. Somit wird der Cursor wenn der Editor vollständig dargestellt wurde, an seiner Ausgangsposition angezeigt.

```

    jsr showcsr
    rts

```

Wenn Sie das Programm SHOWREMOVECSR nun starten, sollte sich am Bildschirm folgendes Bild zeigen:



Da wir den Cursor nun dargestellt haben, wollen wir ihn natürlich auch bewegen können.

Dazu müssen wir das Unterprogramm keyctrl erweitern und wir wollen mit der Bewegung nach rechts beginnen.

Ich habe das Unterprogramm keyctrl ein wenig umgebaut:

```
keyctrl
keyloop      ; tastatur abfragen bis eine
              ; taste gedrueckt wird
              jsr $ffe4
              beq keyloop
              ; cursor nach rechts?
              cmp csrrightkey
              bne check_quit
              jsr csrright
              jmp keyloop
check_quit   ; editor beenden?
              cmp quitkey
              beq keyctrl_end
              ; ansonsten wieder mit der
              ; tastaturabfrage fortsetzen
              jmp keyloop
```

```

keyctrl_end
; vor dem beenden des editors
; noch den bildschirm loeschen
; dies geschieht durch ausgabe
; des zeichens mit dem
; code 147 (clear)

lda scr_code_clrscr
jsr $ffd2

rts

```

Durch den Vergleich CMP csrrightkey wird geprüft, ob der Benutzer die Taste für die Cursorbewegung nach rechts gedrückt hat. Falls nicht, wird zum Label check_quit gesprungen.

Dort wird, wie bereits beschrieben, geprüft, ob der Benutzer die Taste zum Beenden des Editors gedrückt hat und entsprechend reagiert.

Falls der Benutzer jedoch die Taste für die Cursorbewegung nach rechts gedrückt hat, wird das Unterprogramm csrright aufgerufen und anschließend durch den Befehl JMP keyloop wieder mit der Tastaturabfrage fortgesetzt.

Nachfolgend sehen Sie das Unterprogramm csrright:

```

;-----
; csrright
; bewegt den cursor nach rechts
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
;-----
csrright
; pruefen ob cursor bereits
; am rechten rand des editors
; steht

lda csr_col
cmp csr_maxcol

; wenn ja, dann cursor lassen
; wo er ist und zurueck zum
; aufrufer

beq csrright_end

; ansonsten cursor bewegen
; dazu erstmal an der
; aktuellen position entfernen

jsr removecsr

; es geht nach rechts, also
; cursorspalte um 1 erhoehen

inc csr_col

; nun auch die adresse im
; bildschirmspeicher
; aktualisieren und ebenfalls
; um 1 erhoehen

```

```

        ldx #$01
        ldy #$00
        jsr adc16

        ; cursor an neuer position
        ; anzeigen
        jsr showcsr
csrright_end
        rts

```

Hier wird zunächst geprüft, ob der Cursor überhaupt nach rechts bewegt werden kann, denn wenn er sich bereits am rechten Rand des Editors befindet, soll er nicht darüber hinaus bewegt werden können.

Dazu wird die Spalte, in der der Cursor aktuell steht in den Akkumulator geladen und mit dem Inhalt von `csrmaxcol` verglichen. Falls der Vergleich positiv ausfällt, sich der Cursor also bereits am rechten Rand befindet, wird er nicht bewegt und durch den Sprung zum Label `csrright_end` zum Aufrufer zurückgekehrt.

Kann der Cursor jedoch nach rechts bewegt werden, dann wird er zuerst durch den Aufruf von `JSR removecsr` an der aktuellen Position entfernt, sodass das Zeichen, das sich aktuell dort befindet, wieder nicht-revers dargestellt wird.

Im nächsten Schritt wird der Inhalt der Variablen `csrcol` um 1 erhöht, weil der Cursor sich ja um eine Position nach rechts bewegt hat.

Nun wird es interessant.

Wie bereits erwähnt, wird die Adresse im Bildschirmspeicher, an der sich der Cursor aktuell befindet, bei jeder Cursorbewegung aktualisiert. Jedoch nicht durch eine aufwendige Berechnung durch das Unterprogramm `calcposadr`, sondern es reicht eine sehr viel einfachere Addition.

Denn durch die Bewegung nach rechts, wird die Adresse des Cursors um 1 erhöht und dies führen wir nun im nächsten Schritt durch eine 16bit Addition aus.

Wie sie sich erinnern, erwartet das Unterprogramm `adc16` die erste Zahl aufgeteilt auf die Speicherstellen `$FB` und `$FC` und die zweite Zahl aufgeteilt auf das X Register und Y Register.

Na, klingt's warum ich mir für die Speicherung der Cursoradresse ausgerechnet die Speicherstellen `$FB` und `$FC` ausgesucht habe?

Richtig, denn dadurch ist die erste Zahl für die Addition bereits dort, wo sie erwartet wird und ich brauche sie nicht mehr dorthin befördern.

Für den Aufruf von `adc16` ist es nur mehr erforderlich, den Wert 1 an den richtigen Ort zu bringen. In diesem Fall brauchen wir das niederwertige Byte `$01` im X Register und das höherwertige Byte `$00` im Y Register.

Und da das Unterprogramm `adc16` das Ergebnis bereits in den Speicherstellen `$FB` und `$FC` ablegt, ersparen wir uns sogar, dafür zu sorgen, dass die aktualisierte Adresse in diesen Speicherstellen landet.

Sie sehen also, dass man mit etwas Planung und Abstimmung der Unterprogramme aufeinander, eine ganze Menge an Ausführungszeit und Speicher einsparen kann.

Wenn Sie nun das Programm CSRRIGHT starten, werden Sie sehen, dass sich der Cursor nun nach rechts bis zum Rand des Editors bewegen lässt und dass auch der ständige Wechsel zwischen reverser und nicht-reverser Anzeige des Zeichens an der aktuellen Cursorposition funktioniert.

Die Umsetzung der Cursorbewegung in die anderen Richtungen funktioniert exakt nach demselben Schema.

Ich habe die einzelnen Schritte hier noch einmal zusammengefasst und durch die Tabelle im Anschluss wird ersichtlich, welche Grenzwerte zum Zug kommen und welche Variablen verändert werden müssen.

- Prüfen ob der Cursor überhaupt in die jeweilige Richtung bewegt werden kann, er sich also nicht bereits am Rand befindet.
- Falls sich der Cursor bereits am Rand befindet, wird er nicht bewegt und das Unterprogramm kann verlassen werden.
- Andernfalls kann der Cursor bewegt werden. Dazu wird er als erstes durch den Aufruf des Unterprogramms removecsr an der aktuellen Position entfernt, sodass das Zeichen an dieser Position wieder nicht-revers dargestellt wird.
- Als nächstes wird die Änderung an der Position des Cursors durchgeführt (siehe Spalte „Änderung an Position“).
- Nun muss noch die Adresse des Cursors im Bildschirmspeicher entsprechend aktualisiert werden (siehe Spalte „Änderung an Cursoradresse \$FB / \$FC“).
- Cursor durch Aufruf des Unterprogramms showcsr an der neuen Position anzeigen (Zeichen an dieser Position wird revers angezeigt).

Funktion	Variable	Vergleichen mit Grenzwert	Änderung an Position	Änderung an Cursoradresse \$FB / \$FC
Cursor nach rechts	csrcol	csrmaxcol	csrcol + 1	+ 1
Cursor nach unten	csrrow	csrmaxrow	csrrow + 1	+ 40
Cursor nach links	csrcol	csrmincol	csrcol - 1	- 1
Cursor nach oben	csrrow	csrminrow	csrrow - 1	- 40

Möglicherweise ist nicht ganz klar, warum man bei der Cursorbewegung nach unten bzw. oben die Adresse des Cursors um 40 erhöhen bzw. verringern muss.

Der Wert 40 resultiert aus der Bildschirmbreite, denn eine Bildschirmzeile umfasst 40 Zeichen und im Bildschirmspeicher liegen die Adressen zweier untereinander liegender Zeichen daher ebenfalls in diesem Adress-Abstand zueinander.

Bewegt man den Cursor nach unten, dann ist die neue Adresse um 40 höher als die aktuelle Adresse und bei der Cursorbewegung nach oben ist die neue Adresse um 40 niedriger als die aktuelle Adresse.

Bei der Bewegung nach rechts bzw. links beträgt die Differenz jeweils nur 1.

Nachfolgend der Assembler-Code der Unterprogramme `csrdown`, `csrleft` und `csrup`. Wenn Sie sich den Code ansehen, werden Sie bei jedem dieser drei Unterprogramme das Schema erkennen, welches ich vorhin beschrieben habe.

Aus diesem Grund habe ich auch auf die vielen Kommentare verzichtet, wie sie noch im Unterprogramm `csrright` zu sehen waren.

```
-----
:  csrdown
:  bewegt den cursor nach unten
:
:  parameter:
:  keine
:
:  rueckgabewerte:
:  keine
:
:  aendert:
:  a,x,y,status
:
csrdown
    lda  csrrow
    cmp  csrmaxrow
    beq  csrdown_end

    jsr  removecsr

    inc  csrrow

    ldx  #$28
    ldy  #$00
    jsr  adc16

    jsr  showcsr

csrdown_end
    rts
```

```
-----
:  csrleft
:  bewegt den cursor nach links
:
:  parameter:
:  keine
:
:  rueckgabewerte:
:  keine
:
:  aendert:
:  a,x,y,status
:
csrleft
    lda  csrcol
    cmp  csrmincol
    beq  csrleft_end
```

```

        jsr  removecsr
        dec  csrcol
        ldx  #$01
        ldy  #$00
        jsr  sbcl6
        jsr  showcsr
csrleft_end
        rts

```

```

;-----
; csrup
; bewegt den cursor nach oben
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
;
csrup
        lda  csrrow
        cmp  csrminrow
        beq  csrup_end

        jsr  removecsr
        dec  csrrow

        ldx  #$28
        ldy  #$00
        jsr  sbcl6

        jsr  showcsr
csrup_end
        rts

```

Und hier noch die Anpassungen im Unterprogramm keyctrl, damit nun auch die Tasten für die Cursorbewegung nach unten, rechts und oben berücksichtigt werden.

```

keyctrl
keyloop
        ; tastatur abfragen bis eine
        ; taste gedrueckt wird

        jsr  $ffe4
        beq  keyloop

        ; cursor nach rechts?

        cmp  csrrightkey
        bne  check_csrdown

        jsr  csrright
        jmp  keyloop

check_csrdown
        ; cursor nach unten?

        cmp  csrdownkey
        bne  check_csrlf

```



```

        jsr csrdown
        jmp keyloop
check_csrleft
        ; cursor nach links?
        cmp csrleftkey
        bne check_csrup
        jsr csrleft
        jmp keyloop
check_csrup
        ; cursor nach oben?
        cmp csrupkey
        bne check_quit
        jsr csrup
        jmp keyloop
check_quit
        ; editor beenden?
        cmp quitkey
        beq keyctrl_end
        ; ansonsten wieder mit der
        ; tastaturabfrage fortsetzen
        jmp keyloop
keyctrl_end
        ; vor dem beenden des editors
        ; noch den bildschirm loeschen
        ; dies geschieht durch ausgabe
        ; des zeichens mit dem
        ; code 147 (clear)
        lda scrcode_clrscr
        jsr $ffd2
        rts

```

Auch hier lässt sich ein sich wiederholendes Schema erkennen. Es wird nacheinander geprüft, ob der Tastencode der gedrückten Taste einer der von uns definierten Tastencodes entspricht.

Schlägt der Vergleich fehl, wird der Vergleich mit dem nächsten Code fortgesetzt, bis eine Übereinstimmung gefunden wird. In diesem Fall wird dann das entsprechende Unterprogramm aufgerufen und anschließend die Tastaturabfrage durch Sprung zum Label keyloop fortgesetzt.

Findet sich keine Übereinstimmung, so wird ebenfalls wieder mit der Tastaturabfrage fortgesetzt.

Die vollständige Cursorsteuerung habe ich im Programm CSRCTRL umgesetzt. Probieren Sie es am besten im TMP aus und studieren den Code ein wenig wenn Sie wollen.

So, nun können wir unseren Cursor bereits bewegen und sind dadurch bei der Umsetzung des Sprite-Editors einen riesengroßen Schritt weitergekommen.

Als nächstes werden wir uns dem Setzen und Löschen der einzelnen Bits widmen.

Das Setzen eines Bits geschieht durch Drücken der Return-Taste, wohingegen das Löschen eines Bits durch Drücken der Leertaste erfolgt.

In beiden Fällen wird der Cursor um eine Position nach rechts bewegt, damit das Setzen bzw. Löschen von aufeinanderfolgenden Bits rascher von der Hand geht.

Befindet sich der Cursor bereits am rechten Rand des Editors, wird das aktuelle Bit zwar gesetzt oder gelöscht, aber der Cursor verbleibt an dieser Position, da er sich ja nicht über den rechten Rand des Editors hinaus bewegen darf.

Bevor mit der Umsetzung beginnen, müssen wir zunächst im Datenabschnitt am Ende des Programms zwei neue Variablen mit Screencodes ergänzen.

```
scrcode_bit0
    .byte $2e

scrcode_bit1
    .byte $51
```

Ein gelöscht Bit wird durch einen kleinen Punkt (scrcode_bit0) dargestellt, ein gesetztes Bit durch einen großen Punkt (scrcode_bit1).

Zum Setzen und Löschen der Bits habe ich das folgende Unterprogramm namens drawbit geschrieben.

```
-----
; drawbit
; setzt oder loescht das bit an der
; aktuellen cursorposition
;
; parameter:
; carryflag = 0: bit loeschen
; carryflag = 1: bit setzen
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
;
drawbit
    bcc drawbit_bit0
    lda scrcode_bit1
    jmp drawbit_show

drawbit_bit0
    lda scrcode_bit0

drawbit_show
    ldy #$00
    sta ($fb),y

    lda csrcol
    cmp csrmaxcol
    beq drawbit_end

    inc csrcol
    ldx #$01
    ldy #$00
    jsr adc16

drawbit_end
    jsr showcsr
    rts
```

Es hat nur einen Parameter, welcher zur Abwechslung nicht in einem Register übergeben wird, sondern durch den Zustand des Carryflags. Dadurch wird signalisiert, ob wir ein Bit setzen oder löschen wollen.

Zustand des Carryflags	Wirkung
0	Bit an der aktuellen Position wird gelöscht
1	Bit an der aktuellen Position wird gesetzt

Sehen wir uns nun das Unterprogramm drawbit an.

Hier wird als erstes durch die Anweisung BCC drawbit_bit0 geprüft ob das Carryflag gelöscht ist und in diesem Fall zum Label drawbit_bit0 gesprungen.

An dieser Stelle wird der Screencode des kleinen Punktes (gelöschtes Bit), in den Akkumulator geladen.

Ist das Carryflag hingegen gesetzt, wird der Screencode des großen Punktes (gesetztes Bit) in den Akkumulator geladen und dann zum Label drawbit_show gesprungen.

Dort wird durch den Befehl LDY #\$00 der Index für die Adressierung geladen und mit dem nächsten Befehl STA (\$FB),Y der zuvor geladene Screencode an die aktuelle Adresse des Cursors im Bildschirmspeicher geschrieben.

Dort steht nun entweder ein kleiner Punkt (gelöschtes Bit) oder ein großer Punkt (gesetztes Bit).

Der nächste Schritt ist nun die Bewegung des Cursors um eine Position nach rechts.

Zunächst prüfen wir, ob der Cursor bereits am rechten Rand des Editors steht. In diesem Fall kann der Cursor nicht nach rechts bewegt werden und wir springen daher direkt zum Label drawbit_end.

Dort wird das Unterprogramm showcsr aufgerufen, d.h. der Cursor wird an der neuen Position angezeigt. Dies bewirkt, wie bereits bekannt, eine reverse Darstellung des Zeichens an der aktuellen Position. Falls der Cursor bereits am rechten Rand des Editors war, wurde er nicht bewegt und die neue Position entspricht eben wieder der vorherigen Position.

Kann der Cursor jedoch bewegt werden, dann wird die Variable csrcol um 1 erhöht und die aktuelle Adresse des Cursors ebenfalls. Dann gelangen wir zum Label drawbit_end und sind wieder beim Aufruf des Unterprogramms showcsr, wodurch der Cursor an der neuen Position angezeigt wird.

Nun müssen wir nur noch das Unterprogramm keyctrl anpassen, damit die beiden neuen Tastaturfunktionen auch ausgeführt werden.

Hier der entsprechende Ausschnitt aus dem Unterprogramm, welcher den Abschnitt zeigt, den ich für das Setzen und Löschen von Bits hinzugefügt habe:

```

check_csrup
; cursor nach oben?
    cmp csrupkey
    bne check_setbit1
    jsr csrup
    jmp keyloop

check_setbit1
; bit setzen?
    cmp setbit1key
    bne check_setbit0
    sec
    jsr drawbit
    jmp keyloop

check_setbit0
; bit loeschen?
    cmp setbit0key
    bne check_quit
    clc
    jsr drawbit
    jmp keyloop

```

Den aktuellen Stand des Editors finden Sie im Programm DRAWBIT. Probieren Sie es aus und falls Sie bis hierher durchgehalten und alles verstanden haben, können Sie wirklich stolz auf sich sein!

In diesem ersten Teil haben wir extrem viel dazugelernt, z.B. wie man diverse Rechenoperationen im 16bit Bereich umsetzt und die indirekte Y nachindizierte Zeropage-Adressierung.

Aber was noch viel wichtiger ist: Wir haben gesehen, wie die Komponenten zusammenspielen und wie man durch geschickte Organisation Ausführungszeit und Speicher einsparen kann.

Machen Sie mal eine Pause und seien Sie stolz auf das bisher Erreichte.

Im zweiten Teil werden wir dann noch die folgenden restlichen Programmfunktionen umsetzen:

- Speichern des Sprites in einer Datei
- Laden des Sprites aus einer Datei
- Löschen des Editorbereichs
- Anzeige von Hilfsinformationen

Teil 2

Zum aktuellen Zeitpunkt können wir mit unserem Sprite-Editor das Aussehen des Sprites durch Setzen und Löschen von Punkten am Bildschirm definieren.

Das ist schon mal ganz gut, aber leider erst die halbe Miete, denn am Bildschirm allein hilft uns das Punktmuster ja nicht viel. Was wir letztendlich brauchen, sind jene 63 Bytes, die wir dann in den Speicher schreiben können, so wie wir es im Kapitel zum Thema Sprites immer gemacht haben.

Wie wir bereits wissen, ist jedes Zeichen, das wir am Bildschirm sehen, an einer bestimmten Adresse im Bildschirmspeicher zu finden. In dieser Speicherstelle steht dann der Screencode des jeweiligen Zeichens.

Dies gilt klarerweise auch für jene Zeichen, aus denen unser Sprite-Editor besteht.

Das Sprite besteht aus 21 Reihen zu je 3 Bytes und deswegen besteht unsere Editorfläche ebenfalls aus 21 Reihen und 24 Punkten (3 Bytes x 8 Bits pro Byte ergibt 24 Punkte)

Für die nachfolgenden Ausführungen möchte ich das linke Byte als Byte 0, das mittlere Byte als Byte 1 und das rechte Byte als Byte 2 benennen.

Hier das Bitmuster des Ballons, wie es im Editor zu sehen ist:



Und hier als Tabelle mit gesetzten und gelöschten Bits. Ersichtlich ist auch die Aufteilung der Bits auf Byte 0,1 und 2:

	Byte 0								Byte 1								Byte 2							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
00	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0
01	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0
02	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
03	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0
04	0	0	0	0	0	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	0	0
05	0	0	0	0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	0	0	0	0
06	0	0	0	0	0	1	1	1	1	1	0	1	1	0	0	1	1	1	1	1	0	0	0	0
07	0	0	0	0	0	0	1	1	1	1	1	0	0	1	1	1	1	1	1	0	0	0	0	0
08	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
09	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
10	0	0	0	0	0	0	1	0	1	1	1	1	1	1	1	1	0	1	0	0	0	0	0	0
11	0	0	0	0	0	0	0	1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	1	0	0	1	1	1	1	1	0	0	1	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	1	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0

Unsere Aufgabe besteht nun darin, aus den Punktmustern am Bildschirm die entsprechenden Bytewerte herauszurechnen.

Sehen Sie sich beispielsweise Reihe 06 an:



Wenn wir diese 24 Punkte auf die 3 Bytes aufteilen, dann ergeben sich für Reihe 06 folgende Werte:

Byte Nr.	Punktmuster am Bildschirm	Binärer Wert	Hexadezimaler Wert	Dezimaler Wert
0		00000111	\$07	7
1		11011001	\$D9	217
2		11110000	\$F0	240

Wir brauchen also ein Unterprogramm, dem wir eine Reihenummer (0 bis 20) und eine Bytenummer (0,1 oder 2) als Parameter übergeben und das uns daraus die Adresse im Bildschirmspeicher berechnet, an der das entsprechende Punktmuster beginnt.

Wenn wir diese Adresse kennen, können wir aus den 8 Zeichen, die ab dieser Adresse im Bildschirmspeicher liegen, den entsprechenden Bytewert errechnen.

Machen wir uns daher gleich an die Berechnung dieser Adresse.

Dazu müssen wir zunächst jene Adresse im Bildschirmspeicher ermitteln, an der der Beginn einer bestimmten Reihe, genauer gesagt das Zeichen rechts neben der zweiten Ziffer der Reihenummer, zu finden ist.

Wenn wir diese Adresse ermittelt haben, dann haben wir auch gleichzeitig die Adresse, an der das linke Byte (Byte 0) in der Reihe beginnt. 8 Adressen weiter beginnt das mittlere Byte (Byte 1) und wiederum 8 Adressen weiter beginnt das rechte Byte (Byte 2).

Doch eins nach dem anderen.

Um die vorhin genannte Startadresse einer Reihe zu ermitteln, benutzen wir das Unterprogramm `calcposaddr`. Wie wir uns erinnern, übernimmt dieses Unterprogramm einen Zeilen- und Spaltenwert und errechnet daraus die entsprechende Adresse im Bildschirmspeicher.

Doch welchen Zeilen- und Spaltenwert muss man an das Unterprogramm übergeben, um die Startadresse einer Reihe zu erhalten?

Die Angabe des Spaltenwertes ist leicht, denn er ist für jede Reihe gleich und entspricht dem Inhalt der Variablen `csrmincol`.

Diese Variable enthält ja jenen Spaltenwert, den der Cursor bei der Bewegung nach links nicht unterschreiten darf, da er sich ja sonst auf die zweite Ziffer der Reihenummer bewegen würde.

Die Suche nach dem Zeilenwert ist ebenfalls nicht schwierig, da man nichts weiter tun muss als die Reihenummer (diese reicht von 0 bis 20) zum Inhalt der Variablen `csrminrow` zu addieren.

Diese Variable enthält ja jenen Zeilenwert, den der Cursor bei der Bewegung nach oben nicht unterschreiten darf, da er sich ja sonst in den String, welcher am oberen Rand die Bitpositionen darstellt, hineinbewegen würde.

Soweit so gut, wir rufen nun also das Unterprogramm `calcposaddr` mit dem Zeilenwert (`csrminrow + Nummer der Reihe`) und dem Spaltenwert `csrmincol` auf und erhalten wie gewünscht die Adresse im Bildschirmspeicher, welche den Beginn der angegebenen Reihe darstellt.

Doch wie kommen wir nun auf die Adresse des linken, mittleren und rechten Bytes in dieser Reihe?

Ebenfalls ganz einfach, das linke Byte hat die Nummer 0, das mittlere Byte die Nummer 1 und das rechte Byte die Nummer 2.

Wenn wir nun diese Nummer mit 8 multiplizieren erhalten wir den Abstand, den wir zur Startadresse der Reihe addieren müssen, um auf die Startadresse des jeweiligen Bytes zu kommen.

Eine Multiplikation mit 8 lässt sich durch dreimalige Anwendung des Befehls `ASL` erreichen. Dadurch wird der Wert um drei Bitpositionen nach links verschoben und dies entspricht einer Multiplikation mit 8.

Bytenummer	Abstand zur Startadresse der Reihe
0	$0 * 8 = 0$
1	$1 * 8 = 8$
2	$2 * 8 = 16$

Nun werden wir dies alles in ein Unterprogramm namens `calcbyteaddr` verpacken.

Es übernimmt zwei Parameter, erstens die Nummer der Reihe (reicht von 0 bis 20) und zweitens die Nummer des gewünschten Bytes (0,1 oder 2)

Als Ergebnis erhalten wir dann die Adresse, an der das Punktmuster für das jeweilige Byte im Bildschirmspeicher beginnt.

Nachfolgend sehen Sie den Assemblercode des Unterprogramms:

```

:-----:
: calcbyteaddr
: berechnet an welcher adresse im
: bildschirmspeicher ein bestimmtes
: byte des spriterasters beginnt
:
: parameter:
: reihennr (0-20): y register
: bytenr (0,1 oder 2): x register
:
: rueckgabewerte:
: adresse: lo/hi in $fd/$fe
:
: aendert:
: a,x,y,status
:
calcbyteaddr
; speicherstellen $fb und $fc
; auf dem stack sichern
    lda $fb
    pha
    lda $fc
    pha
; als parameter uebergebene
; bytenummer ebenfalls auf
; dem stack sichern
    txa
    pha
; uebergebene reihennummer in
; den akku kopieren
    tya
; nun den inhalt der variablen
; csrminrow hinzuaddieren
    clc
    adc csrminrow
; summe als parameter fuer
; calcposadr ins y register
; kopieren (zeilenwert)
    tay

```



```

; der parameter fuer den
; spaltenwert entspricht
; dem inhalt der variablen
; csrmincol

ldx csrmincol

; adresse berechnen
; niederwertiges byte steht
; dann im x register und das
; hoeherwertige byte im
; y register

jsr calcposaddr

; bytenummer wieder vom stack
; holen

pla

; bytenummer mit 8
; multiplizieren, ergibt den
; adressindex fuer das
; gewaehlte byte

asl a
asl a
asl a

; diesen index zu der vorhin
; berechneten startadresse
; der reihe hinzuaddieren

sta $fb

lda #$00
sta $fc

jsr adcl6

; nun haben wir die adresse
; des gewuenschten bytes
; aufgeteilt auf die
; speicherstellen $fb und $fc
; diese speicherstellen werden
; jedoch fuer die adresse
; des cursors verwendet und
; daher kopieren wir die werte
; in die speicherstellen
; $fd und $fe um

lda $fb
sta $fd

lda $fc
sta $fe

; gesicherten inhalte der
; speicherstellen $fb und $fc
; wieder vom stack holen
; und wiederherstellen

pla
sta $fc

pla
sta $fb

rts

```

Zu Beginn werden hier die Inhalte der Speicherstellen \$FB und \$FC sowie auch der Inhalt des X Registers (welches die als Parameter übergebene Bytenummer enthält) auf dem Stack gesichert, weil diese innerhalb des Unterprogramms verändert werden.

Nun wird, wie vorhin beschrieben, der Zeilenwert für den Aufruf des Unterprogramms calcposaddr gebildet.

Dazu wird die im Y Register als Parameter übergebene Reihenummer in den Akkumulator kopiert und durch den Befehl ADC csrminrow der Zeilenwert berechnet (Reihenummer + csrminrow).

Dieser errechnete Zeilenwert wird dann in das Y Register kopiert, weil ihn das Unterprogramm calcposaddr dort als Parameter erwartet.

Nun wird noch der Spaltenwert, also der Inhalt der Variablen csrmincol, in das X Register geladen und das Unterprogramm calcposaddr aufgerufen.

Als Ergebnis erhalten wir das niederwertige Byte der Startadresse der Reihe im X Register und das höherwertige Byte im Y Register.

Nun holen wir uns die zuvor auf dem Stack gesicherte Bytenummer wieder vom Stack und multiplizieren diese durch dreimalige Anwendung des Befehls ASL mit 8.

Diesen Wert müssen wir für die 16bit Addition nun in die beiden Speicherstellen \$FB und \$FC kopieren. Die zweite Zahl für die 16bit Addition (Startadresse der Reihe) befindet sich ja bereits aufgeteilt auf das X Register und das Y Register.

Das Ergebnis dieser Addition ist nun unser Endergebnis, also die Startadresse des gewünschten Bytes.

Sie befindet sich nun aufgeteilt auf die Speicherstellen \$FB und \$FC. Diese Speicherstellen können wir jedoch nicht zur Rückgabe des Endergebnisses nutzen, da sie ja für die laufende Speicherung der Cursoradresse verwendet werden.

Daher kopieren wir die beiden Werte in die ungenutzten Speicherstellen \$FD und \$FE um.

Nun müssen wir nur noch die gesicherten Inhalte der Speicherstellen \$FB und \$FC wieder vom Stack holen und wiederherstellen.

Nachdem dies geschehen ist, können wir mit dem Befehl RTS zum Aufrufer zurückkehren.

Den aktuellen Stand des Programms habe ich unter dem Namen CALCBYTEADDR auf der Diskette zur Verfügung gestellt.

Den Hauptteil am Beginn des Programms habe ich vorübergehend etwas geändert, damit wir die korrekte Funktion des neuen Unterprogramms überprüfen können.

```

;-----
; hauptprogramm
; hier wird das programm gestartet
; start von basic aus mit sys 12288
;
;*= $3000
;
; sichern der parameter,
; welche vom basic programm im
; x register und y register
; abgelegt wurden, weil sie
; bereits durch den aufruf des
; unterprogramms init
; veraendert werden
;
txa
pha

tya
pha
; editor variablen
; initialisieren

jsr init

; benutzeroberflaeche anzeigen
; jsr draweditor
; tastaturabfrage starten
; jsr keyctrl

; unterprogramm calcbyteaddr
; testen

; zuvor gesicherte parameter
; wieder vom stack holen und
; inhalte der register x und y
; wiederherstellen

pla
tay

pla
tax

; adresse berechnen

jsr calcbyteaddr

rts

```

Die Aufrufe der Unterprogramme draweditor und keyctrl habe ich auskommentiert, da sie für den Test nicht gebraucht werden. Den Aufruf des Unterprogramms init brauchen wir jedoch, da dieses unter anderem für die richtige Initialisierung der Variablen csrminrow und csrmincol verantwortlich ist.

Nachdem wir das Programm assembliert haben, wechseln wir zu Basic, geben NEW ein und laden das Basic-Programm BYTEADDRTEST von der Diskette.

```

LOAD"BYTEADDRTEST",8
SEARCHING FOR BYTEADDRTEST
LOADING
READY.
LIST

10 REM REIHENNUMMER SETZEN
20 POKE 782,20
30 REM BYTENUMMER SETZEN
40 POKE 781,2
50 REM TESTPROGRAMM AUFRUFEN
60 SYS 12288
70 REM ERRECHNETE ADRESSE AUSGEBEN
80 PRINT PEEK(253)+256*PEEK(254)
READY.
RUN
1889
READY.

```

Mit diesem Programm kann man durch Angabe der Reihennummer und Bytenummer auf einfache Art und Weise überprüfen, ob uns das Unterprogramm calcbyteaddr die richtige Adresse liefert.

In Zeile 20 wird die Reihennummer in die Speicherstelle 782 und die Bytenummer in die Speicherstelle 781 geschrieben.

Durch den Befehl SYS 12288 in Zeile 60 werden dadurch die Reihennummer in das Y Register und die Bytenummer in das X Register geladen.

Die beiden Werte landen also genau dort, wo Sie unser Unterprogramm calcbyteaddr erwartet.

Nachdem das Unterprogramms durchgelaufen ist, steht uns die Adresse aufgeteilt auf die Speicherstellen \$FD und \$FE zur Verfügung.

In Zeile 80 wird die Adresse aus den Inhalten dieser beiden Speicherstellen errechnet und ausgegeben, sodass wir sie auf Korrektheit überprüfen können.

Im obigen Beispiel haben wir als Parameter die Reihe Nr. 20 und das Byte Nr. 2 angegeben.

Als Ergebnis erhalten wir die Adresse 1889 und wollen nun nachrechnen, ob dieser Wert korrekt ist.

Basierend auf den Inhalten der Variablen editorrow und editorcol enthält die Variable csrminrow den Wert 1 und die Variable csrmincol den Wert 9.

Daraus ergeben sich folgende Parameter für das Unterprogramm calcposaddr:

Zeilenwert: $\text{csrminrow} + \text{nummer der reihe} = 1 + 20 = 21$

Spaltenwert: $\text{csrmincol} = 9$

Wie uns bereits vom Unterprogramm `calcposaddr` her bekannt ist, berechnen wir nun die Startadresse der Reihe Nr. 20 durch die Formel

$$\text{Zeilenwert} * 40 + \text{Spalte} + \text{Startadresse des Bildschirmspeichers}$$

Dies bedeutet in diesem Beispiel hier:

$$21 * 40 + 9 + 1024 = 1873 \text{ (Startadresse von Reihe Nr. 20)}$$

Nun müssen wir noch den Adressindex für das Byte Nr. 2 hinzuaddieren:

$$1873 + 16 = 1889$$

Passt also!

Experimentieren Sie ruhig ein wenig mit den unterschiedlichsten Reihen- und Bytenummern und vergleichen Sie die ausgegebene Adresse mit jener Adresse, die Sie selbst nach dem obigen Schema errechnet haben.

Gut, nun können wir also durch Angabe einer Reihen- und Bytenummer die Adresse im Bildschirmspeicher ermitteln, an der das entsprechende Punktmuster beginnt.

Der nächste Schritt besteht darin, ein Unterprogramm zu erstellen, welches die 8 Zeichen ab dieser Adresse durchläuft und daraus den entsprechenden Bytewert herausrechnet.

Erinnern Sie sich noch wie man eine Binärzahl in eine Dezimalzahl umrechnet?

Richtig, man addiert die Zweierpotenzen an jenen Stellen, an denen eine 1 steht.


Hier als Beispiel die Umrechnung der binären Zahl 11011001 in eine Dezimalzahl.

Bitposition	7	6	5	4	3	2	1	0
Zweierpotenz	128	64	32	16	8	4	2	1
	1	1	0	1	1	0	0	1

Nun addieren wir die Zweierpotenzen an jenen Stellen, an denen eine 1 vorhanden ist, also

$$128 + 64 + 16 + 8 + 1 = 217$$

Ich habe diese Binärzahl nicht zufällig gewählt, sondern ich habe dazu das Byte Nr. 1 aus der Tabelle weiter oben entnommen, als ich die drei Bytes der Reihe 06 als Beispiel herangezogen habe.

Auf dieselbe Art und Weise gehen wir vor, wenn wir das Punktmuster  in ein Byte umrechnen wollen.

Wir addieren auch hier alle Zweierpotenzen, an denen ein großer Punkt vorhanden ist, denn der große Punkt steht ja für eine 1.

Daher legen wir im Datenabschnitt am Ende des Programms ein Feld namens powersoftwo an, welches die benötigten Zweierpotenzen 128, 64, 32, 16, 8, 4, 2 und 1 enthält.

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

Die Reihenfolge ist hier wichtig, da wir mit der Aufsummierung beim ersten Zeichen von links beginnen und dieses dem höchstwertigen Bit, also jenem mit der Wertigkeit 128 entspricht.

Zusätzlich brauchen wir noch eine Variable namens bytevalue, in der das Unterprogramm das errechnete Ergebnis ablegen kann.

```
bytevalue
    .byte $00
```

Nachfolgend sehen Sie den Assemblercode des Unterprogramms getbytefromscr:

```
-----
: getbytefromscr
: rechnet ein achtstelliges punktmuster
: aus dem editorbereich in ein byte um
:
: parameter:
: adresse: lo/hi in $fd/$fe
:
: rueckgabewerte:
: bytewert in der variablen bytevalue
:
: aendert:
: a,y,status
:
getbytefromscr
    ; y register und variable
    ; bytevalue mit dem wert 0
    ; initialisieren
    ldy #$00
    sty bytevalue
:
calcloop
    ; screencode aus aktueller
    ; adresse im
    ; bildschirmspeicher lesen
    lda ($fd),y
    ;
    ; entspricht der screencode
    ; dem screencode des Zeichens
    ; welches wir fuer die
    ; darstellung einr 1 definiert
    ; haben?
    cmp scrcode_bit1
    ;
    ; wenn nicht, dann weiter
    ; zum naechsten zeichen
    bne calc_continue
    ;
    ; wenn ja dann zweierpotenz
    ; fuer aktuelle position holen
    ; und zum inhalt der variablen
    ; bytevalue addieren
```

```

        lda powersoftwo,y
        clc
        adc bytevalue
        sta bytevalue
calc_continue
        ; index auf naechstes zeichen
        ; setzen
        iny

        ; haben wir bereits alle acht
        ; zeichen durchlaufen?
        cpy #$08

        ; wenn nicht, dann naechstes
        ; zeichen pruefen
        bne calcloop

        ; ansonsten sind wir fertig
        rts

```

Zu Beginn wird das Y Register und die Variable bytevalue mit dem Wert 0 initialisiert.

Das Y Register dient bei beiden indizierten Adressierungen als Index-Register:

Erstens beim Laden des Screencodes von der aktuellen Position in den Akkumulator:

```
lda ($fd),y
```

Und zweitens beim Laden der zur Bitposition passenden Zweierpotenz:

```
lda powersoftwo,y
```

Durch den Befehl LDA (\$FD),y wird der Screenccode des jeweiligen Zeichens in den Akkumulator geladen.

Durch den Befehl CMP scrcode_bit1 wird dieser Screencode mit dem Screencode jenes Zeichens verglichen, das wir als Zeichen zur Darstellung der 1 definiert haben, also mit dem Screencode des großen Punktes.

Schlägt der Vergleich fehl, dann befindet sich an der aktuellen Position der Screencode des kleinen Punktes, also jenes Zeichens, das wir zur Darstellung der 0 definiert haben. Eine 0 können wir immer überspringen und springen daher zum Label calc_continue.

Dort wird durch den Befehl INY der Inhalt des Y Registers um 1 erhöht, d.h. der Index verweist nun auf das nächste Zeichen.

Nun wird noch geprüft ob das Y Register nach dem Erhöhen den Wert 8 enthält. Wenn ja, dann haben wir bereits alle Zeichen an den Indizes 0 bis 7 durchlaufen und die Aufsummierung ist beendet.

Falls nicht, wird das nächste Zeichen geprüft.

Soweit zum Ablauf wenn wir auf eine 0 treffen. Treffen wir jedoch auf eine 1, also auf den Screencode des großen Punktes, dann wird aus dem Feld powersoftwo die Zweierpotenz für diese Position ausgelesen.

Durch die indizierte Adressierung über das Y Register wird hier immer der passende Wert zur der aktuellen Bitposition ausgelesen.

Die ausgelesene Zweierpotenz wird in den Akkumulator geladen, der aktuelle Inhalt der Variablen bytevalue hinzuaddiert und das Ergebnis wieder in die Variable bytevalue zurückgeschrieben.

Auf diese Weise werden alle Zweierpotenzen jener Bitpositionen aufsummiert, an denen eine 1, also der Screencode des großen Punktes, steht.

Zum besseren Verständnis, wie hier der Index im Y Register zum Einsatz kommt, habe ich folgende Tabelle erstellt.

Nehmen wir als Beispiel wieder das Punktmuster in Reihe 6, Byte Nr. 1:



Das Unterprogramm calcbyteaddr liefert uns hier die Adresse 1321 (\$0529) zurück.

Um das Unterprogramm getbytefromscr aufrufen zu können, muss diese Adresse mit dem niederwertigen Byte in der Speicherstelle \$FD und dem höherwertigen Byte in der Speicherstelle \$FE stehen.

Ich habe das Unterprogramm calcbyteaddr so geschrieben, dass es die Adresse in genau diesen beiden Speicherstellen ablegt und es daher auf die optimale Zusammenarbeit mit dem Unterprogramm getbytefromscr ausgerichtet.

Zu Beginn wird das Y Register und die Variable bytevalue mit dem Wert 0 initialisiert und die Adresse wurde wie vorhin erwähnt mit 1321 berechnet.

Hier zum leichteren Vergleich nochmals das Punktmuster:



Die grün hinterlegten Zeilen heben nochmals die Bitpositionen hervor, an denen eine 1 zu finden ist.

Y Register	Adresse + Inhalt des Y Registers	Screencode an dieser Adresse	Entspricht Screencode des Zeichens für „1“ ?	Zweierpotenz (powersoftwo + Inhalt des Y Registers)	Inhalt der Variablen bytevalue
0	$1321 + 0 = 1321$	\$51 (großer Punkt)	JA	128	$0 + 128 = 128$
1	$1321 + 1 = 1322$	\$51 (großer Punkt)	JA	64	$128 + 64 = 192$

Y Register	Adresse + Inhalt des Y Registers	Screencode an dieser Adresse	Entspricht Screencode des Zeichens für „1“ ?	Zweierpotenz (powersoftwo + Inhalt des Y Registers)	Inhalt der Variablen bytevalue
2	1321 + 2 = 1323	\$2E (kleiner Punkt)	NEIN	32	bleibt bei 192
3	1321 + 3 = 1324	\$51 (großer Punkt)	JA	16	192 + 16 = 208
4	1321 + 4 = 1325	\$51 (großer Punkt)	JA	8	208 + 8 = 216
5	1321 + 5 = 1326	\$2E (kleiner Punkt)	NEIN	4	bleibt bei 216
6	1321 + 6 = 1327	\$2E (kleiner Punkt)	NEIN	2	bleibt bei 216
7	1321 + 7 = 1328	\$51 (großer Punkt)	JA	1	216 + 1 = 217

Um die Umrechnung eines Punktmusters in ein Byte zu testen, habe ich das Programm GETBYTETEST auf der Diskette zur Verfügung gestellt.

In das Unterprogramm keyctrl musste ich einen zusätzlichen Codeabschnitt für den Test einbauen.

```

check_getbyte
; test fuer getbytefromscr?
    cmp savekey
    bne check_quit
; reihennummer = 0
    ldy #$00
; bytenummer = 0
    ldx #$00
; adresse des punktmusters
; berechnen
    jsr calcbyteaddr
; cursor ausblenden
    jsr removecsr
; punktmuster in bytewert
; umrechnen
    jsr getbytefromscr
; cursor wieder einblenden
    jsr showcsr
; errechneten wert in
; x register kopieren
    ldx bytevalue
; zurueck zu basic
; ergebnis kann mit

```

```

; print peek (781) geprueft
; werden
rts

```

Da der Code für den Test ja nicht dauerhaft im Programm verbleibt und ich daher auch keine eigene Tastenkombination zu dessen Aufruf definieren wollte, habe ich kurzerhand die Tastenkombination verwendet, die dann später zum Speichern des Sprites in einer Datei verwendet wird (SHIFT + S).

Um den Test so einfach wie möglich zu halten, habe ich die Reihen- und Bytenummer fix auf die Nummer 0 festgelegt.

Was hier auffällt ist, dass der Cursor vom Bildschirm entfernt wird. Warum ist das nötig? Der Grund dafür ist folgender: Wenn der Cursor an irgendeiner Position im Punktmuster steht, dann steht an dieser Position im Bildschirmspeicher der Screencode des reversen Zeichens.

Das Unterprogramm getbytefromscr würde daher das Zeichen an dieser Stelle nicht erkennen, da es ja nur auf den Screencode des großen Punktes prüft.

Aus diesem Grund wird der Cursor ausgeblendet, bevor mit dem Auslesen des Punktmusters begonnen wird. Sobald dies beendet ist, wird der Cursor wieder eingeblendet.

Letztendlich wird der errechnete Bytewert in das X Register kopiert, damit er von Basic aus leicht mit dem Befehl PRINT PEEK(781) ausgelesen werden kann.

Nachdem Sie den Editor mit SYS 12288 gestartet haben, erzeugen Sie mit der Return- und der Leertaste im linken Byte der Reihe 00 ein beliebiges Punktmuster. Wichtig ist es, die Reihe 00 und das Byte Nr. 0 zu verwenden, da das Testprogramm wie vorhin erwähnt, darauf festgelegt ist.

Notieren Sie sich das von Ihnen erstellte Punktmuster.

Durch Drücken der Tastenkombination SHIFT + S findet ein Wechsel zu Basic statt und sie können sich den errechneten Bytewert durch Eingabe des Befehls PRINT PEEK (781) ausgeben lassen.

Nun wandeln Sie diesen Bytewert mit einem Taschenrechner ins Binärsystem um und prüfen ob das Bitmuster mit dem von Ihnen erzeugten Punktmuster übereinstimmt. Wenn ja, ist alles gut gegangen!


Spielen wir diesen Test mal anhand des vorhin verwendeten Punktmusters durch.

Starten Sie den Editor mit SYS 12288 und erzeugen Sie das folgende Punktmuster:



Wenn Sie nun die Tastenkombination SHIFT + S drücken, findet der Wechsel zu Basic statt und der Befehl PRINT PEEK(781) sollte Ihnen nun den Bytewert des Punktmusters ausgeben, welches Sie erzeugt haben.

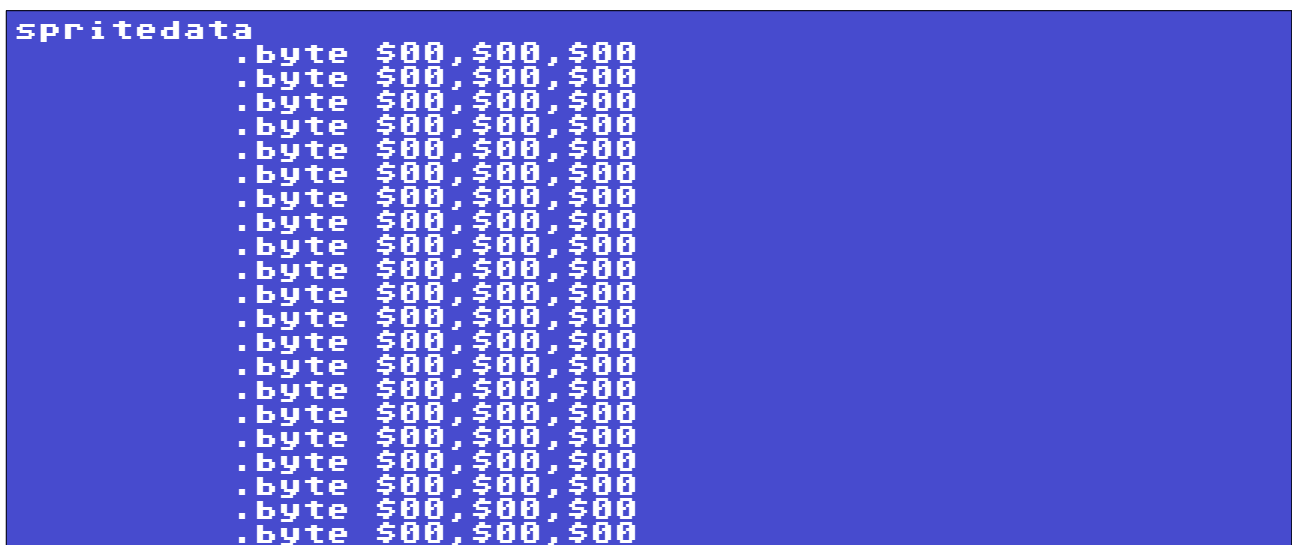


Wie wir hier sehen, stimmt das Ergebnis, denn der dezimale Wert 217 entspricht der Binärzahl %11011001, was sich auch mit dem Punktmuster  deckt.

So, nun brauchen wir nur noch ein einziges Unterprogramm, um das komplette Punktmuster im Editor vom Bildschirm zu lesen.

Doch zunächst müssen wir erst mal klären, wo wir die 63 Bytes, die wir vom Bildschirm lesen, überhaupt speichern wollen.

Dazu legen wir uns im Datenabschnitt ein Feld namens `spritedata` bestehend aus 63 Bytewerten an.



In diesem Feld werden wir die 63 Bytes, die wir aus dem Punktmuster im Editor errechnet haben, ablegen.

Dazu erstellen wir ein Unterprogramm namens sprfromscreen, welches den Editorbereich Reihe für Reihe, Byte für Byte durchläuft und die dabei errechneten Bytes in dieses Feld schreibt.

Zusätzlich brauchen wir noch drei weitere Variablen als Zähler innerhalb des Unterprogramms.

```
rownr      .byte $00
bytenr     .byte $00
spritedataindex
           .byte $00
```

Hier das Unterprogramm sprfromscreen:

```
-----
: sprfromscr
: wandelt die punktmuster im editor
: in bytewerte um und legt diese im
: feld spritedata ab
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,y,status
:
sprfromscr
; wir beginnen bei reihe nr. 0
    lda #$00
    sta rownr

; und schreiben die bytes
; ab index 0 in das feld
; spritedata
    sta spritedataindex

rowloop
; in jeder reihe beginnen wir
; mit byte nr. 0
    lda #$00
    sta bytenr

byteloop
; adresse fuer aktuelles
; byte-punktmuster berechnen
    ldy rownr
    ldx bytenr
    jsr calcbyteaddr

; punktmuster in bytewert
; umrechnen
    jsr getbytefromscr

; errechneten wert ins
; feld spritedata schreiben
```

```

lda bytevalue
ldx spritedataindex
sta spritedata,x

; index fuer spritedata
; erhoehen

inc spritedataindex
; bytenummer erhoehen
inc bytenr

; schon alle bytes in dieser
; reihe bearbeitet?

lda bytenr
cmp #$03

; wenn nicht dann mit
; naechstem byte weitermachen

bne byteloop

; ansonsten weiter zur
; naechsten reihe

; reihennummer erhoehen
inc rownr

; haben wir schon alle reihen
; ausgelesen?

lda rownr
cmp #$15

; wenn nicht, dann weiter zur
; naechsten reihe

bne rowloop

; ansonsten sind wir fertig

rts

```

Im Unterprogramm werden die Punktmuster in zwei verschachtelten Schleifen ausgelesen, in Bytewerte umgerechnet und diese im Datenfeld spritedata abgelegt.

Die äußere Schleife (rowloop) durchläuft die Reihen mit der Zählervariablen rownr und die innere Schleife (byteloop) durchläuft die Byte-Punktmuster in jeder Reihe mit der Zählervariablen bytenr.

Die innere Schleife wird solange durchlaufen, bis der Inhalt der Variablen bytenr nach dem Erhöhen den Wert 3 erreicht hat, wohingegen die äußere Schleife solange durchlaufen wird, bis alle Reihen abgearbeitet wurden. Dies ist dann der Fall, wenn der Inhalt der Variablen rownr nach dem Erhöhen den Wert 21 annimmt.

Während die Schleifen durchlaufen werden, wird für das jeweilige Punktmuster, das durch rownr und bytenr gegeben ist, zunächst die Adresse durch den Aufruf des Unterprogramms calcbyteaddr ermittelt.

Diese geht dann direkt weiter an das Unterprogramm getbytefromscr, welches das errechnete Byte in der Variablen bytevalue ablegt.

Dieser Wert wird dann im Feld spritedata an der jeweiligen Position, welche durch den Index bestimmt wird, abgelegt. Der Index wird, während die beiden Schleifen durchlaufen werden, von 0 bis 62 hochgezählt.

Im Unterprogramm keyctrl war folgende Ergänzung nötig:

```
check_save
; sprite speichern?
    cmp savekey
    bne check_quit
    jsr savesprite
    rts
```

Drückt der Benutzer die Tastenkombination SHIFT + S, dann wird das Unterprogramm savesprite aufgerufen.

Nachfolgend sehen Sie das Unterprogramm savesprite. Im Kommentarblock steht, dass das Sprite in einer Datei gespeichert wird. Soweit sind wir jedoch noch nicht, aber ich habe diese Info trotzdem schon mal dort angegeben, weil dies im Falle eines positiven Ergebnisses des Testprogramms unser nächster Schritt ist.

```
-----
; savesprite
; uebertraegt das punktmuster aus dem
; editor in das datenfeld spritedata
; und speichert diese 63 bytes in einer
; datei
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
savesprite
; cursor ausblenden
    jsr removecsr

; punktmuster ins feld
; spritedata uebertragen
    jsr sprfromscr

; cursor wieder einblenden
    jsr showcsr

; adresse des feldes
; spritedata im x register
; und y register ablegen
; damit das feld von basic
; aus ausgelesen werden kann

    ldx #<spritedata
    ldy #>spritedata

    rts
```

Hier wird zunächst der Cursor aus dem erwähnten Grund ausgeblendet. Dann wird das Punktmuster aus dem Editor in das Datenfeld spritedata übertragen und der Cursor wieder eingeblendet.

Anschließend wird das X Register mit dem niederwertigen Byte der Adresse von spritedata und das Y Register mit der höherwertigen Adresse von spritedata geladen.

Dadurch ist diese Adresse von Basic aus zugänglich und wir können das Datenfeld auslesen.

Als nächstes wollen wir nun testen, ob die Punktmuster aus dem Editor korrekt in die entsprechenden Bytewerte umgerechnet werden.

Ich habe das Test-Programm unter dem Namen SPRFROMSCRTEST auf der Diskette zur Verfügung gestellt. Laden Sie das Programm und starten den Sprite-Editor.

Nun erstellen Sie bitte im Sprite-Editor das Punktmuster für den Ballon und wenn Sie die Erstellung abgeschlossen haben, drücken Sie die Tastenkombination SHIFT + S, was zu einem Wechsel nach Basic führt.

Dort geben Sie NEW ein, laden das Basic-Programm SPRITEBYTES.



```
19.....  
20.....  
PRESS SHIFT + H FOR HELP  
READY.  
NEW  
READY.  
LOAD"SPRITEBYTES",8  
SEARCHING FOR SPRITEBYTES  
LOADING  
READY.  
LIST  
10 PRINT CHR$(147)  
20 A=PEEK(781)+256*PEEK(782)  
30 FOR R=0 TO 20  
40 FOR B=0 TO 2  
50 PRINT PEEK(A+R*3+B);  
60 NEXT B  
70 PRINT  
80 NEXT R  
READY.
```

Das Programm löscht zunächst den Bildschirm und liest den Inhalt des X Registers und des Y Registers aus den Speicherstellen 781 und 782. Das Unterprogramm savesprite hat dort vor seiner Beendigung das niederwertige und höherwertige Byte der Adresse des Feldes spritedata abgelegt.

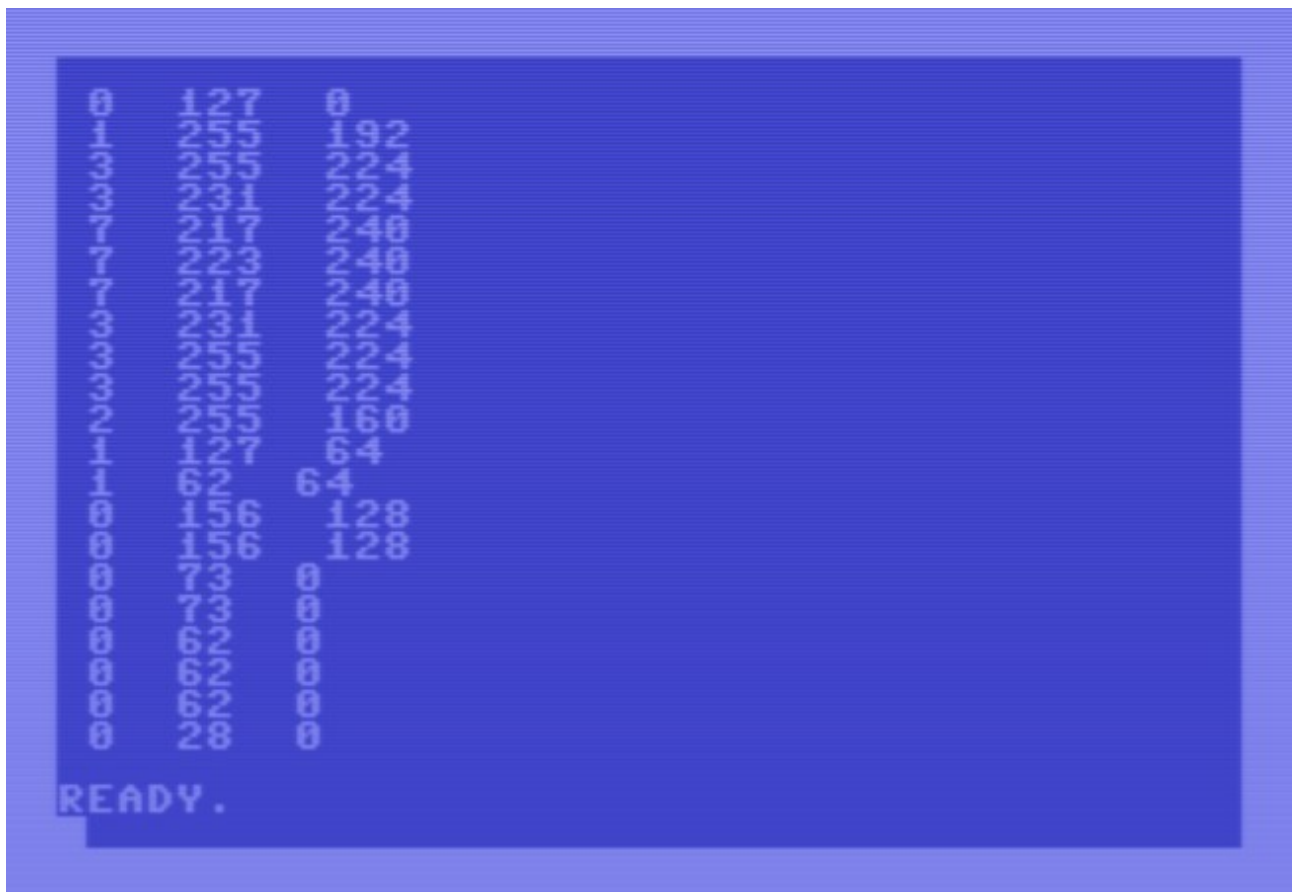
In Zeile 20 wird aus diesen beiden Werten wieder die Adresse gebildet und in der Variablen A gespeichert.

Nun hat das Basic-Programm direkten Zugriff auf das Feld spritedata.

Es liest die 63 Bytes aus dem Feld und gibt diese in 21 Reihen zu je 3 Bytes am Bildschirm aus.

Wenn Sie das Programm nun mit RUN starten, sollten Sie folgende Ausgabe am Bildschirm sehen.

Und wenn die Werte (so wie hier) auch noch mit den Werten in der darauffolgenden Tabelle identisch sind, dann hat alles korrekt funktioniert :)



Byte Nr. 0	Byte Nr. 1	Byte Nr. 2
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

Dateioperationen in Assembler

Da wir unsere Spritedaten in eine Datei speichern bzw. auch wieder aus dieser laden wollen, müssen wir uns damit beschäftigen, wie man diese Vorgänge in Assembler umsetzt.

Ich habe daher ein Programm namens SPRITEIO erstellt, welches Ihnen, losgelöst vom Sprite-Editor, genau dies zeigen soll.

Es besteht aus folgenden Unterprogrammen:

- **sprdatatofile:** Speichert die Bytes, die sich im Datenfeld spritedata befinden, in eine Datei namens SPRITE.
- **scratchfile:** Löscht die Datei namens SPRITE.
- **sprdatafromfile:** Lädt die Bytes wieder aus der Datei namens SPRITE und speichert diese im Datenfeld spritedata.
- **clrspriedata:** Setzt alle Bytes innerhalb des Datenfelds spritedata auf den Wert 0.
- **showsprite:** Zeigt das Sprite, dessen Aussehen durch die Bytes im Datenfeld spritedata definiert wird, am Bildschirm an.

Am Ende des Programms befindet sich ein Datenabschnitt mit folgenden Variablen:

```
spritedata
    .byte $00,$7f,$00
    .byte $01,$ff,$c0
    .byte $03,$ff,$e0
    .byte $03,$e7,$e0
    .byte $07,$d9,$f0
    .byte $07,$df,$f0
    .byte $07,$d9,$f0
    .byte $03,$e7,$e0
    .byte $03,$ff,$e0
    .byte $03,$ff,$e0
    .byte $02,$ff,$a0
    .byte $01,$7f,$40
    .byte $01,$3e,$40
    .byte $00,$9c,$80
    .byte $00,$9c,$80
    .byte $00,$49,$00
    .byte $00,$49,$00
    .byte $00,$3e,$00
    .byte $00,$3e,$00
    .byte $00,$3e,$00
    .byte $00,$1c,$00

filename
    .text "sprite,seq"

scratchfilename
    .text "s:sprite"
```

Wie Sie sehen, enthält das Datenfeld spritedata nicht lauter Nullbytes, sondern ich habe hier, um den Anschluss zum vorherigen Beispiel nicht zu verlieren, die Daten des Ballons eingetragen.

Die Variable filename enthält den Dateinamen, unter dem die Spritedaten abgespeichert werden und die Variable scratchfilename dient als Floppy-Befehl zum Löschen der Datei.

Das Programm führt nun folgende Schritte aus:

- Die Datei SPRITE wird gelöscht
- Die Bytes aus dem Datenfeld spritedata werden in die Datei SPRITE geschrieben
- Die Bytes werden wieder aus der Datei SPRITE in das Datenfeld spritedata eingelesen
- Das Sprite wird am Bildschirm angezeigt

Bei Dateioperationen kann natürlich so einiges schiefgehen, z.B. dass eine zu lesende oder zu löschende Datei gar nicht existiert, dass ein Schreibvorgang, aus welchen Gründen auch immer, fehlschlägt usw.

Was genau schiefgegangen ist, kann man dann beispielsweise über den Fehlerkanal der Floppy auslesen und entsprechend reagieren. Das Thema Fehlerbehandlung werden wir im Anschluss noch behandeln.

Für den Moment möchte ich mich eher auf das Wesentliche konzentrieren, nämlich wie man die Spritedaten in eine Datei schreibt und diese nachher wieder aus dieser Datei einlesen kann.

Die Dateioperationen werden über eine Reihe von Kernal-Funktionen aufgerufen:

Name	Adresse	Zweck
SETLFS	\$FFBA	Fileparameter setzen
SETNAM	\$FFBD	Filenamen setzen
OPEN	\$FFC0	Öffnen der Datei
CHKOUT	\$FFC9	Default Output ändern
PRINT	\$FFD2	Zeichen in Datei schreiben
CLRCH	\$FFCC	Default Output / Input wieder zurücksetzen
CLOSE	\$FFC3	Schließen der Datei
CHKIN	\$FFC6	Default Input ändern
INPUT	\$FFCF	Zeichen lesen

Beginnen wir mit dem ersten Punkt der Liste, also dem Löschen der Datei namens SPRITE.

In Basic würde das so aussehen:

```
OPEN 1,8,15,"S:SPRITE":CLOSE 1
```

Und hier die Umsetzung in das Assembler-Unterprogramm scratchfile:

```
-----
: scratchfile
: loescht die datei namens sprite
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,y,status
:
scratchfile
; setfls
lda #$01
```

```

ldx #$08
ldy #$0f
jsr $ffba
; setnam

lda #$08
ldx #<scratchfilename
ldy #>scratchfilename
jsr $ffbd

; open
jsr $ffc0

; close

lda #$01
jsr $ffc3

rts

```

Zuallererst wird immer die Kernal-Funktion SETLFS aufgerufen.

Über diese Funktion werden jene Fileparameter gesetzt, die in obiger Basic-Zeile gelb markiert sind, also die logische Dateinummer, die Geräteadresse und die Sekundäradresse.

Die logische Filenummer kommt in den Akkumulator, die Geräteadresse ins X Register und die Sekundäradresse ins Y Register.

Als nächstes folgt immer der Aufruf der Kernal-Funktion SETNAM. Dadurch gibt man jenen String an, der in obiger Basic-Zeile grün markiert ist. In diesem Fall hier ist dies der Inhalt der Variablen scratchfilename, also S:SPRITE.

Die Länge des Strings (in diesem Fall 8) kommt in den Akkumulator und die Adresse des Strings übergibt man aufgeteilt auf das X Register und das Y Register, wobei das niederwertige Byte der Adresse im X Register und das höherwertige Byte der Adresse im Y Register stehen muss.

Darauf folgt der Aufruf der Funktion OPEN, also der türkis markierte Teil in obiger Basic-Zeile. Diese Funktion hat keine Parameter, da alle Informationen bereits durch die beiden vorherigen Funktionen eingestellt wurden.

Das ist auch der Grund, warum vor einem Aufruf von OPEN immer ein Aufruf der Funktionen SETFLS und SETNAM erfolgen muss.

Im Falle von Ein- und Ausgaben würden nun Aufrufe von weiteren Kernal-Funktionen (z.B. PRINT oder INPUT) folgen, aber da es hier um das Löschen der Datei geht, fehlt nur noch der orange markierte Teil in der in obigen Basic-Zeile, also der Aufruf der Funktion CLOSE.

Die Funktion CLOSE benötigt für die Ausführung nur die logische Dateinummer im Akkumulator.

Soweit so gut, nehmen wir uns daher das Schreiben der Spritedaten in die Datei vor.

Sehen Sie sich dazu das Assembler-Unterprogramm sprdatatofile an:

```
-----
: sprdatatofile
: schreibt die bytes aus dem datenfeld
: spritedata in eine sequentielle datei
: namens sprite
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,y,status
:
sprdatatofile
; setfls
    lda #$01
    ldx #$08
    ldy #$01
    jsr $ffba
; setnam
    lda #$0a
    ldx #<filename
    ldy #>filename
    jsr $ffbd
; open
    jsr $ffc0
; chkout
    ldx #$01
    jsr $ffc9
; bytes in datei schreiben
    ldx #$00
writeloop
; byte aus spritedata holen
    lda spritedata,x
; byte in datei schreiben
    jsr $ffd2
; index auf naechstes byte
    inx
; bereits alle bytes
; geschrieben?
    cpx #$3f
; falls nicht, dann weiter
; zum naechsten byte
    bne writeloop
; clrch
    jsr $ffcc
; close
    lda #$01
    jsr $ffc3
```

Die Aufrufe von SETFLS, SETNAM und OPEN entsprechen folgendem Basic-Befehl:

OPEN 1,8,1,“SPRITE,SEQ“

Nun müssen wir den Ausgabekanal auf unsere Datei umleiten. Dies geschieht mittels der Kernal-Funktion CHKOUT.

Sie übernimmt im X Register die logische Dateinummer jener Datei, in die Ausgaben geschrieben werden sollen.

Da wir für unsere Datei die logische Dateinummer 1 vergeben haben, tragen wir diese Nummer auch vor dem Aufruf von CHKOUT im X Register ein.

Wir haben die Funktion CHROUT (\$FFD2) schon öfters für die Ausgabe von Zeichen auf dem Bildschirm aufgerufen. Sie wird ebenfalls genutzt, um Zeichen in eine Datei zu schreiben.

In diesem Fall ist jedoch ein vorheriger Aufruf der Funktion CHKOUT nötig, um über die Angabe der logischen Dateinummer die Datei auszuwählen, in die diese Zeichen geschrieben werden sollen.

Ab diesem Zeitpunkt landen alle über den Aufruf der Funktion CHROUT ausgegebenen Zeichen nicht mehr am Bildschirm, sondern in dieser Datei.

Nun werden in einer Schleife namens writeloop nacheinander alle Bytes des Datenfeldes spritedata in den Akkumulator geladen. Da sich die Funktion CHROUT den Zeichencode des zu schreibenden Zeichens aus dem Akkumulator holt, kann sie darauffolgend direkt ohne Umschweife aufgerufen werden, wodurch das jeweilige Zeichen in der Datei landet.

Danach wird der Index durch den Befehl INX auf das nächste Byte gesetzt und geprüft, ob schon alle Bytes aus dem Datenfeld spritedata verarbeitet wurden. Falls dies nicht der Fall sein sollte, wird wieder zum Label writeloop gesprungen und das nächste Byte verarbeitet.

Ansonsten wird durch den Aufruf der Kernal-Funktion CLRCH der Ausgabekanal wieder auf den Bildschirm eingestellt, wodurch die Ausgaben ab nun nicht mehr in die Datei, sondern wieder auf dem Bildschirm ausgegeben werden.

Zum Schluss wird die Datei noch geschlossen, was durch den Aufruf der Funktion CLOSE und vorherigem Laden der logischen Dateinummer in den Akkumulator erfolgt.

So, nun haben wir unsere Spritedaten in die Datei geschrieben und nun wollen wir sie von dort wieder in das Datenfeld spritedata holen.

Hier sehen Sie den Assemblercode des Unterprogramms sprdatafromfile:

```
-----  
: sprdatafromfile  
: laedt die spritedaten aus einer  
: sequentiellen datei namens sprite  
: in das datenfeld spritedata  
:  
: parameter:  
: keine  
:  
: rueckgabewerte:  
: keine  
:  
: aendert:  
: a,x,y,status  
:  
sprdatafromfile  
    ; setfls  
    lda #$01  
    ldx #$08  
    ldy #$00  
    jsr $ffba  
    ; setnam  
    lda #$0a  
    ldx #<filename  
    ldy #>filename  
    jsr $ffbd  
    ; open  
    jsr $ffc0  
    ; chkin  
    ldx #$01  
    jsr $ffc6  
    ; bytes aus datei lesen  
    ldx #$00  
readloop  
    ; byte aus datei lesen  
    jsr $ffcf  
    ; gelesenes byte in feld  
    ; spritedata schreiben  
    sta spritedata,x  
    ; index auf naechstes byte  
    inx  
    ; bereits alle bytes gelesen?  
    cpx #$3f  
    ; falls nicht, dann weiter  
    ; zum naechsten byte  
    bne readloop  
    ; clrch  
    jsr $ffcc  
    ; close  
    lda #$01  
    jsr $ffc3
```

Die Aufrufe von SETFLS, SETNAM und OPEN entsprechen folgendem Basic-Befehl:

OPEN 1,8,0,"SPRITE,SEQ"

Analog zu vorhin, müssen wir als nächstes unsere Datei als Eingabekanal festlegen, da normalerweise Eingaben ja von der Tastatur gelesen werden und nicht aus einer Datei.

Dies geschieht durch Aufruf der Kernal-Funktion CHKIN, welche die logische Dateinummer im X Register erwartet.

Von nun an wird nicht mehr von der Tastatur, sondern aus unserer Datei gelesen. Das gelesene Byte wird an die durch den aktuellen Index bestimmte Stelle im Datenfeld spritedata geschrieben.

Dann wird der Index durch den Befehl INX auf das nächste Byte gesetzt und geprüft, ob bereits alle 63 Bytes aus der Datei gelesen wurden. Falls dies nicht der Fall sein sollte, wird wieder zum Label readloop gesprungen und das nächste Byte verarbeitet.

Ansonsten wird durch den Aufruf der Kernal-Funktion CLRCH der Eingabekanal wieder auf die Tastatur eingestellt, wodurch die Eingaben ab nun nicht mehr aus der Datei, sondern wieder von der Tastatur entgegengenommen werden.

Hinweis:

Die Funktion CLRCH setzt sowohl den Standard-Ausgabekanal als auch den Standard-Eingabekanal zurück auf den Bildschirm bzw. die Tastatur.

Zum Schluss wird wie vorhin die Datei noch geschlossen, was durch den Aufruf der Funktion CLOSE und vorherigem Laden der logischen Dateinummer in den Akkumulator erfolgt.

Das Unterprogramm showsprite ist nur hier in diesem Programm von Bedeutung. Es erzeugt aus den Spritedaten im Datenfeld spritedata ein Sprite und stellt es am Bildschirm dar. In diesem Fall wird in der linken oberen Ecke ein gelber Ballon angezeigt. Auf diese Weise können wir kontrollieren, ob alle Bytes korrekt geschrieben und gelesen wurden.

```

:-----
: showsprite
: zeigt das sprite dessen aussehen
: durch das feld spritedata definiert
: wird am bildschirm an
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,y,status
:
showsprite
: block 11 fuer das sprite
: auswaehlen

```



```

        lda #$0b
        sta $07f8

        ; bytes aus dem datenfeld
        ; spritedata in diesen block
        ; kopieren
        ldx #$00
copyloop
        ; byte kopieren
        lda spritedata,x
        sta $02c0,x

        ; index auf naechstes byte
        inx

        ; bereits alle bytes kopiert?
        cpx #$3f

        ; falls nicht, dann weiter
        ; zum naechsten byte
        bne copyloop

        ; sprite ist einfarbig
        lda #$00
        sta $d01c

        ; sprite soll gelb sein
        lda #$07
        sta $d027

        ; spriteposition festlegen
        ; x = $18 (24), y = $32 (50)
        lda #$18
        sta $d000

        lda #$32
        sta $d001

        ; x ist <= 255, daher setzen
        ; wir das erweiterungsbit
        ; fuer die x koordinate
        ; zurueck
        lda #$00
        sta $d010

        ; sprite hat prioritaeet vor
        ; dem hintergrund
        lda #$00
        sta $d01b

        ; sprite aktivieren
        lda #$01
        sta $d015

        rts

```

Hier begegnet uns nichts neues, denn all dies kennen wir bereits aus dem Kapitel zum Thema Sprites.

Was jedoch zu erwähnen wäre:

Da wir es hier in diesem Programm ja sowieso nur mit einem einzigen Sprite, dem Sprite 0, zu tun haben, habe ich auf das Maskieren der Bits beim Setzen der Werte verzichtet.

Sehen Sie z.B. den letzten Abschnitt oben an:

```
        ; sprite aktivieren
        lda #$01
        sta $d015
        rts
```

Dieser Befehl aktiviert Sprite 0 und deaktiviert gleichzeitig die Sprites 1 bis 7. Da wir aber nur mit Sprite 0 arbeiten, ist es für uns nicht von Bedeutung, dass die Sprites 1 bis 7 deaktiviert werden.

Ein anderes Beispiel ist dieser Abschnitt hier:

```
        ; sprite ist einfarbig
        lda #$00
        sta $d01c
```

Auch hier wird keine Rücksicht auf die Sprites 1 bis 7 genommen, sondern es werden alle 8 Sprites als einfarbig definiert. Dies ist jedoch für uns nicht von Interesse, da wir ja nur mit dem Sprite 0 arbeiten.

Werfen wir abschließend noch einen Blick auf den Hauptteil des Programms.

```
;-----
; hauptprogramm
; hier wird das programm gestartet
; start von basic aus mit sys 12288
        *= $3000
        ; sprite ausblenden
        lda #$00
        sta $d015
        ; datei loeschen
        jsr scratchfile
        ; spritedaten in datei
        ; schreiben
        jsr sprdatatofile
        ; spritedaten loeschen
        jsr clrspriedata
        ; spritedaten aus datei
        ; laden
        jsr sprdatafromfile
        ; sprite anzeigen
        jsr showsprite
        ; adresse von spritedata
```

```
; im x register und y register  
; hinterlegen, damit das feld  
; von basic aus ausgelesen  
; werden kann  
  
ldx #<spritedata  
ldy #>spritedata  
  
rts
```

Zu Beginn wird das Sprite 0 (und auch alle anderen, was jedoch hier wie gesagt nicht von Bedeutung ist) ausgeblendet, denn es kann ja sein, dass das Sprite durch einen vorausgegangenen Aufruf des Programms noch am Bildschirm angezeigt wird.

Dann wird die Datei namens SPRITE gelöscht. Dann werden die Bytes aus dem Datenfeld spritedata in die neu angelegte Datei gleichen Namens gespeichert.

Nun werden die 63 Bytes aus der Datei namens SPRITE in das Datenfeld spritedata eingelesen und durch der Aufruf des Unterprogramms showsprite erzeugt das Sprite basierend auf diesen Daten und zeigt es am Bildschirm an.

Abschließend wird die noch das niederwertige Byte der Adresse des Datenfeldes spritedata in das X Register und das höherwertige Byte der Adresse in das Y Register geschrieben.

Dadurch hat man die Möglichkeit, von Basic aus auf die Spritedaten zuzugreifen und kann sich diese eventuell nochmal zur Kontrolle ausgeben lassen.

Nachdem Sie das Programm SPRITEIO durch Eingabe von SYS 12288 gestartet haben, sollten Sie nach einigen Sekunden folgendes am Bildschirm sehen:



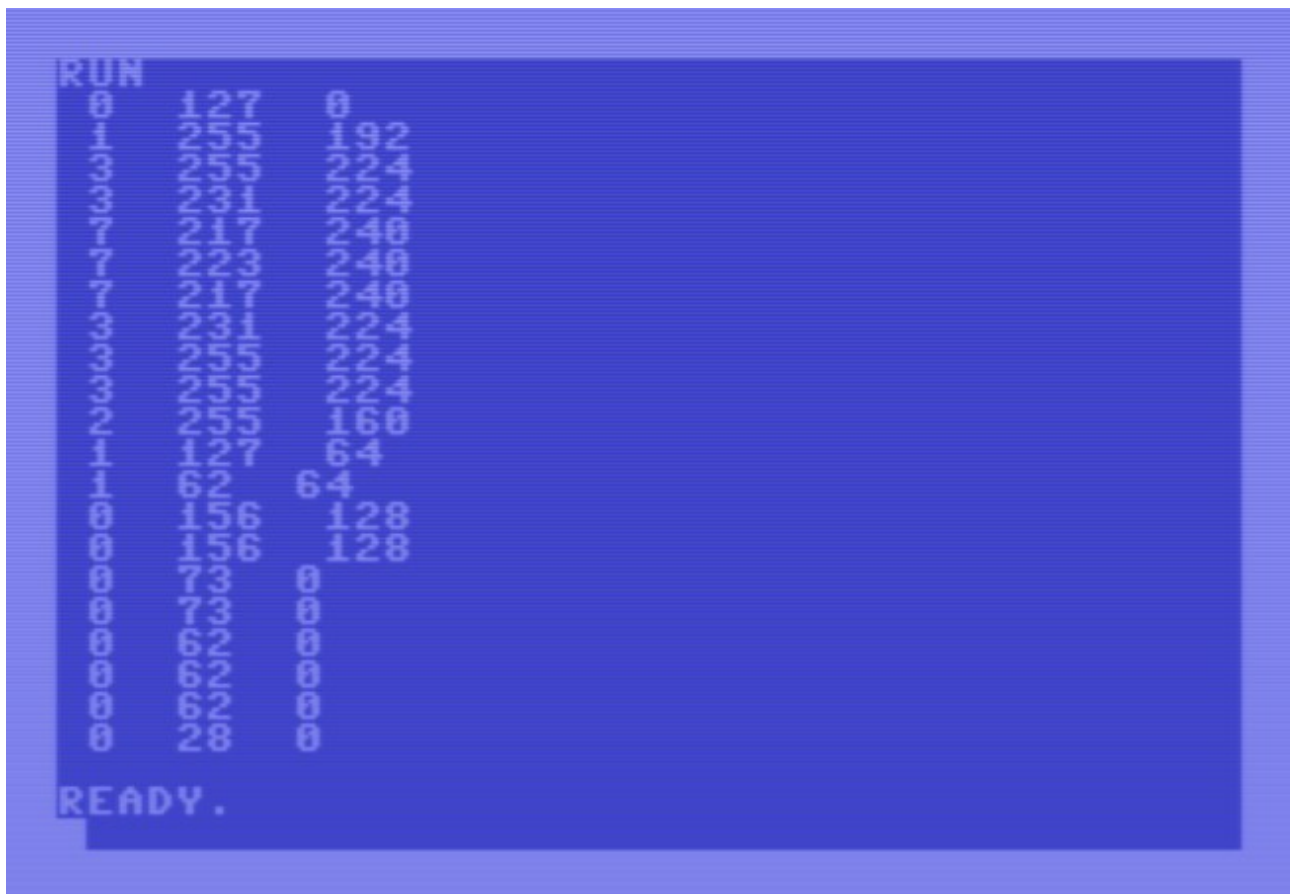
Blenden Sie nun das Sprite durch Eingabe des Befehls POKE 53269,0 aus und laden das Basic-Programm SHOWSPRBYTES.

```
READY.  
SYS 12288  
  
READY.  
POKE 53269,0  
  
READY.  
LOAD"SHOWSPRBYTES",8  
  
SEARCHING FOR SHOWSPRBYTES  
LOADING  
READY.  
LIST  
  
10 A=PEEK(781)+256*PEEK(782)  
20 FOR R=0 TO 20  
30 FOR B=0 TO 2  
40 PRINT PEEK(A+R*3+B);  
50 NEXT B  
60 PRINT  
70 NEXT R  
READY.
```

Das Programm bildet aus dem nieder- und höherwertigen Byte der Adresse des Datenfeldes `spritedata` wieder die volle Adresse und speichert diese in der Variablen `A`.

Dann werden die 63 Bytes aus dem Datenfeld `spritedata` in zwei ineinander verschachtelten Schleifen ausgelesen und in Gruppen zu je 3 Bytes untereinander ausgegeben.

Nach Eingabe von `RUN` sollten nun die 21 Reihen mit jeweils 3 Bytes angezeigt werden.



Ok, soviel zu den elementaren Dateioperationen in Assembler.

Sehen wir uns nun an, wie ich die vorgestellten Unterprogramme zum Speichern und Laden der Spritedaten in den Sprite-Editor integriert habe.

Die Unterprogramme scratchfile, sprdatatofile und sprdatafromfile existieren 1:1 identisch auch im Sprite-Editor.

Damit das Speichern und Laden der Spritedaten über die entsprechenden Tastenkombinationen auch registriert wird, muss das Unterprogramm keyctrl um folgenden Abschnitt ergänzt werden:

```
check_save
    ; sprite speichern?
    cmp savekey
    bne check_load
    jsr savesprite
    jmp keyloop

check_load
    ; sprite laden?
    cmp loadkey
    bne check_quit
    jsr loadsprite
    jmp keyloop
```

Und die Abfrage unmittelbar davor muss ebenfalls an das neue Label check_save angepasst werden:

```
check_setbit0
; bit loeschen?
    cmp setbit0key
    bne check_save

    clc
    jsr drawbit
    jmp keyloop
```

Neu sind auch die beiden Unterprogramme savesprite und loadsprite:

```
-----
; savesprite
; uebertraegt das punktmuster aus dem
; editor in das datenfeld spritedata
; und speichert diese 63 bytes in einer
; datei
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
savesprite
; rahmenfarbe fuer die dauer
; des speichervorgangs aendern
    lda #$0d
    sta $d020

    ; cursor ausblenden
    jsr removecsr

    ; punktmuster ins feld
    ; spritedata uebertragen
    jsr sprfromscr

    ; cursor wieder einblenden
    jsr showcsr

    ; datei loeschen
    jsr scratchfile

    ; spritedaten in datei
    ; schreiben
    jsr sprdatatofile

    ; rahmenfarbe wieder auf
    ; vorherige farbe einstellen
    lda #$0e
    sta $d020

    rts
```

Hier erkennt man die einzelnen Schritte wieder, die beim Speichern der Spritedaten durchlaufen wurden. Neu hinzugekommen ist jedoch die Änderung der Rahmenfarbe für die Dauer des Speichervorgangs.

Um dem Benutzer optisch anzuzeigen, dass die Spritedaten gerade in die Datei geschrieben werden, ändert sich die Rahmenfarbe nach dem Drücken der Tastenkombination SHIFT + S.

Der Rahmen wird solange in dieser Farbe angezeigt, bis der Speichervorgang abgeschlossen ist und wenn dies der Fall ist, erscheint der Rahmen wieder in der vorherigen Farbe.

Das Unterprogramm loadsprite ist noch nicht fertig, es werden hier zum jetzigen Zeitpunkt nur die Spritedaten aus der Datei in das Datenfeld spritedata geladen.

```
-----
: loadsprite
: laedt die spritedaten aus der datei
: namens sprite ins datenfeld
: spritedata und gibt diese wieder
: als punktmuster am bildschirm aus
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,y,status
:
loadsprite
    ; spritedaten aus datei laden
    jsr sprdatafromfile
    rts
```

Angenommen, wir haben mit dem aktuellen Stand des Sprite-Editors ein Sprite erstellt, dieses in der Datei SPRITE gespeichert und wollen dessen Aussehen irgendwann nun ändern.

Dazu laden wir mit dem Unterprogramm sprdatafromfile die Spritedaten aus der Datei in das Datenfeld spritedata.

Das Problem: Dort können wir sie nicht sehen, d.h. wir müssen nun den umgekehrten Weg beschreiten und die Bytes aus dem Datenfeld wieder in Punktmuster umwandeln und im Sprite-Editor anzeigen.

Nehmen wir an, dass sich an irgendeiner Stelle im Datenfeld spritedata der Wert 217 befindet.

Dies entspricht dem binären Wert %11011001, was am Bildschirm dem Punktmuster



entspricht.

Wir brauchen also nun ein Unterprogramm, das uns ausgehend von einem Byte aus dem Datenfeld spritedata wieder das entsprechende Punktmuster auf dem Bildschirm bringt.

Doch bevor wir ein Punktmuster am Bildschirm ausgeben können, müssen wir zuerst einmal wissen, wo wir es ausgeben müssen, besser gesagt an welcher Adresse im Bildschirmspeicher.

In den 8 Speicherstellen ab dieser Startadresse wird dann das Punktmuster Zeichen für Zeichen im Bildschirmspeicher wieder aufgebaut.

Falls in unserem Byte, dessen Punktmuster wir wieder auf den Bildschirm bringen wollen, ein Bit gesetzt ist, schreiben wir an den entsprechenden Positionen den Screencode für den großen Punkt in den Bildschirmspeicher und andernfalls den Screencode für den kleinen Punkt.

Aber wie prüfen wir, ob in dem Byte ein Bit an einer bestimmten Position gesetzt ist?

Dies lässt sich mit einer UND-Verknüpfung zwischen dem Byte und der Zweierpotenz, welche dieser Bitposition entspricht, herausfinden.

Nehmen wir als Beispiel wieder das genannte Byte mit dem Inhalt %11011001.

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

Die erste Zeile enthält die Nummern der Bitpositionen, die zweite Zeile die jeweilige Zweierpotenz an dieser Position und die dritte Zeile die Bits unserer Zahl, wobei die Stellen an denen eine 1 steht, grün markiert sind.

Um die UND-Verknüpfungen nach der Reihe für jede Bitposition durchführen zu können, legen wir uns im Datenabschnitt ein Feld mit den benötigten Zweierpotenzen an. Wichtig ist hierbei die Reihenfolge, wir beginnen bei 128 (\$80) und enden bei 1 (\$01).

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

Spielen wir den Aufbau des Punktmusters im Bildschirmspeicher mal anhand eines Beispiels losgelöst vom Assembler-Code durch.

Angenommen, wir zählen im Y Register einen Index von 0 bis 7 hoch. Dieser Index soll sowohl als Index in das Datenfeld powersoftwo als auch als Index für die indirekte Y nachindizierte Adressierung bezogen auf die Startadresse des Punktmusters im Bildschirmspeicher dienen.

Vorhin habe ich erwähnt, das wir zuerst diese Startadresse ausrechnen müssen, bevor wir mit der Ausgabe des Punktmusters beginnen können.

Nehmen wir an, diese Berechnung hätte die Adresse 1100 (\$044C) ergeben.

Y Register	Zweierpotenz	UND-Verknüpfung	Ergebnis der UND-Verknüpfung	Adresse im Bildschirmspeicher (1100),Y	Auszugebendes Zeichen
0	powersoftwo,0 =128	%11011001 %10000000	%10000000	1100 + 0 = 1100	■
1	powersoftwo,1 =64	%11011001 %01000000	%01000000	1100 + 1 = 1101	■
2	powersoftwo,2 =32	%11011001 %00100000	%00000000	1100 + 2 = 1102	■
3	powersoftwo,3 =16	%11011001 %00010000	%00010000	1100 + 3 = 1103	■
4	powersoftwo,4 =8	%11011001	%00001000	1100 + 4 = 1104	■

Y Register	Zweierpotenz	UND-Verknüpfung	Ergebnis der UND-Verknüpfung	Adresse im Bildschirmspeicher (1100),Y	Auszugebendes Zeichen
		%00001000			
5	powersoftwo,5 =4	%11011001	%00000000	1100 + 5 = 1105	•
		%00000100			
6	powersoftwo,6 =2	%11011001	%00000000	1100 + 6 = 1106	•
		%00000010			
7	powersoftwo,7 =1	%11011001	%00000001	1100 + 7 = 1107	•
		%00000001			

Und hier die Umsetzung des in der Tabelle dargestellten Ablaufs in Assembler-Code:

```

;-----
; putbytetoscr
; wandelt ein byte in ein achstelliges
; punktmuster um und gibt dieses in
; der uebergebenen reihe und "byte" des
; sprites aus
;
; parameter:
; bytewert: in variablen bytevalue
; reihennr (0 bis 20): y register
; bytenr (0,1 oder 2): x register
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
;
putbytetoscr
; adresse des punktmusters im
; bildschirmspeicher berechnen
    jsr calcbyteaddr
    ldy #$00
;
; printloop
    lda powersoftwo,y
    ; bit gesetzt?
    and bytevalue
    beq bit_notset
;
; bit_set
    lda scrcode_bit1
    jmp print_continue
;
; bit_notset
    lda scrcode_bit0
;
; print_continue
    sta ($fd),y
    iny
    cpy #$08
    bne printloop
;
    rts

```

Zuerst wird anhand der übergebenen Reihen- und Bytenummer die Adresse berechnet, an der das entsprechende Punktmuster im Bildschirmspeicher beginnt.

Nun beginnt der Ablauf, welcher in der obigen Tabelle dargestellt ist.

Die zur Bitposition passende Zweierpotenz wird aus dem Datenfeld powersoftwo in den Akkumulator geladen.

Das Byte, welches als Punktmuster dargestellt werden soll, wurde in der Variablen bytevalue als Parameter an das Unterprogramm übergeben.

Nun wird durch die UND-Verknüpfung festgestellt, ob das jeweilige Bit gesetzt ist oder nicht.

Wenn nicht, wird zum Label bit_notset gesprungen, an dem der Screencode für den kleinen Punkt in den Akkumulator geladen wird.

Wenn ja, wird der Screencode für den großen Punkt in den Akkumulator geladen.

Nun wird der Screencode durch den Befehl STA (\$FD),Y an die jeweilige Position im Bildschirmspeicher geschrieben.

Dies wiederholt sich für alle acht Bitpositionen bis schlussendlich das vollständige, achtstellige Punktmuster am Bildschirm zu sehen ist.

Mit dem Unterprogramm putbytetoscr kann man also Bytes in Punktmuster umwandeln und basierend auf der Reihen- und Bytenummer auch an der korrekten Position am Bildschirm darstellen.

Was nun noch zur Ausgabe fehlt, ist ein Unterprogramm, welches Byte für Byte das komplette Datenfeld spritedata durchläuft, in Punktmuster umwandelt und diese eines nach dem anderen am Bildschirm ausgibt.

Wir nennen es sprtoscr und hier sehen Sie dessen Assembler-Code:

```
-----  
: sprtoscr  
: wandelt die bytes im datenfeld  
: spritedata in die entsprechenden  
: punktmuster auf dem bildschirm um  
:  
: parameter:  
: keine  
:  
: rueckgabewerte:  
: keine  
:  
: aendert:  
: a,x,y,status  
:  
sprtoscr  
    lda #$00  
    sta rownr  
    sta spritedataindex  
  
row_loop  
    lda #$00  
    sta bytenr  
  
byte_loop  
    ldx spritedataindex  
    lda spritedata,x  
    sta bytevalue  
  
    ldx bytenr  
    ldy rownr
```

```

        jsr  putbytetoscr
        inc  spritedataindex
        inc  bytenr
        lda  bytenr
        cmp  #$03
        bne  byte_loop

        inc  rownr
        lda  rownr
        cmp  #$15
        bne  row_loop

        rts

```

Das Unterprogramm durchläuft das Datenfeld spritedata Byte für Byte und ruft für jedes Byte das zuvor erstellte Unterprogramm putbytetoscr auf.

Die Variable rownr enthält dabei die jeweilige Reihenummer und die Variable bytenr die jeweilige Bytenummer für den Aufruf von putbytetoscr.

Die Variable spritedataindex wird während des Ablaufs des Unterprogramms sprtoscr von 0 bis 62 hochgezählt und dient als Index im X Register, um sich mit dem Befehl LDA SPRITEDATA,X ein Byte nach dem anderen aus dem Datenfeld spritedata zu holen.

Es sind hier zwei ineinander verschachtelte Schleifen zu erkennen. Eine äußere Schleife, welche über wiederholtes Springen zum Label row_loop die Reihen 0 bis 20 abarbeitet und eine innere Schleife, welche die drei Bytes 0 bis 2 in jeder Reihe bearbeitet und über wiederholtes Springen zum Label byte_loop durchlaufen wird.

Als letzten Schritt müssen wir noch den Cursor an die Ausgangsposition, also in die linke obere Ecke des Editorbereichs versetzen.

Dazu habe ich folgendes kleines Unterprogramm namens csrhome geschrieben:

```

-----
: csrhome
: setzt den cursor an die
: ausgangsposition, also in die linke
: obere ecke des editorbereichs
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,y,status
csrhome
        ldx  csrhomeaddr
        ldy  csrhomeaddr+1
        stx  $fb
        sty  $fc

        lda  csrminrow
        sta  csrrow

        lda  csrmincol
        sta  csrcol

        jsr  showcsr

```

Beim Starten des Editors wurde in der Variablen `csrhomeaddr` jene Adresse im Bildschirmspeicher hinterlegt, die der Ausgangsposition des Cursors, also der linken oberen Ecke des Editorbereichs entspricht.

Das Unterprogramm `csrhome` tut nun nichts anderes, als genau diese Adresse in die Speicherstellen `$FB` und `$FC` zu transportieren und die Variablen `csrrow` und `csrcol` ebenfalls wieder mit jenen Werten zu versorgen, welche den Koordinaten der Ausgangsposition des Cursors entsprechen.

Nun können wir das Unterprogramm `loadsprite` vervollständigen:

```

-----
loadsprite
; laedt die spritedaten aus der datei
; namens sprite ins datenfeld
; spritedata und gibt diese wieder
; als punktmuster am bildschirm aus
;
parameter:
keine
;
rueckgabewerte:
keine
;
aendert:
a,x,y,status
loadsprite
; rahmenfarbe fuer die dauer
; des ladevorgangs aendern
lda #$0d
sta $d020
; spritedaten aus datei laden
jsr sprdatafromfile
; spritedaten in punktmuster
; am bildschirm umwandeln
jsr sprtoscr
; cursor an die
; ausgangsposition setzen
jsr csrhome
; rahmenfarbe wieder auf
; vorherige farbe einstellen
lda #$0e
sta $d020
rts

```

Analog zum Speichern der Spritedaten ändert sich auch während des Ladens der Spritedaten die Rahmenfarbe, um damit anzuzeigen, dass der Ladevorgang aktuell im Gange ist.

Nachdem die Spritedaten aus der Datei ins Datenfeld `spritedata` transportiert wurden, werden sie durch den Aufruf des Unterprogramms `sprtoscr` wieder in Punktmuster auf dem Bildschirm verwandelt.

Dann wird der Cursor in die linke obere Ecke des Editorbereichs versetzt und der Rahmen wieder in der ursprünglichen Farbe angezeigt.

Nun fehlt uns nur noch eine einzige Editorfunktion, nämlich die Möglichkeit zum Löschen des Editorbereichs.

Dazu habe ich zunächst das Unterprogramm `clrspriedata` geschrieben:

```
;-----  
; clrspriedata  
; setzt alle bytes im datenfeld  
; spritedata auf den wert 0, loescht  
; also das spritemuster im speicher  
;  
; parameter:  
; keine  
;  
; rueckgabewerte:  
; keine  
;  
; aendert:  
; a,x,status  
clrspriedata  
    lda #$00  
    ldx #$00  
  
clrloop  
    sta spritedata,x  
    inx  
    cpx #$3f  
    bne clrloop  
  
    rts
```

Wie sie hier sehen können, durchläuft es in einer Schleife alle 63 Bytes des Datenfeldes `spritedata` und setzt diese auf den Wert 0.

Den gesamten Vorgang des Löschens verfrachten wir in das Unterprogramm `clrsprite`:

```
;-----  
; clrsprite  
; loescht den editorbereich  
;  
; parameter:  
; keine  
;  
; rueckgabewerte:  
; keine  
;  
; aendert:  
; a,x,status  
clrsprite  
    ; spritedaten loeschen  
    jsr clrspriedata  
    ; leeres sprite im editor  
    ; anzeigen  
    jsr sprtoscr  
    ; cursor an die  
    ; ausgangsposition setzen  
    jsr csrhome  
    rts
```

Durch den Aufruf des soeben definierten Unterprogramms `clrspriedata` werden die Bytes des Datenfelds `spritedata` auf den Wert 0 gesetzt.

Dann werden alle 63 Bytes (welche nun den Wert 0 enthalten) in Punktmuster auf den Bildschirm verwandelt, wodurch wieder eine leere Editorfläche bestehend aus kleinen Punkten angezeigt wird.

Anschließend wird der Cursor noch an die Ausgangsposition versetzt und damit haben wir auch diese Programmfunktion erfolgreich umgesetzt.

Nun müssen wir nur noch den entsprechenden Aufruf in das Unterprogramm `keyctrl` einbauen.

```
check_load
    ; sprite laden?
    cmp loadkey
    bne check_clear
    jsr loadsprite
    jmp keyloop
check_clear
    ; sprite loeschen?
    cmp clearkey
    bne check_quit
    jsr clrsprite
    jmp keyloop
```

Im Abschnitt nach dem Label `check_load` müssen wir nun anstelle des Labels `check_quit` das Label `check_clear` anspringen. Hat der Benutzer die Tastenkombination SHIFT + CLR/HOME gedrückt, wird durch den Befehl `JSR CLRSPRITE` das Unterprogramm `clrsprite` aufgerufen und der Editorbereich gelöscht.

Kommen wir nun zur allerletzten Programmfunktion, der Anzeige eines Fensters mit Hilfsinformationen, welche über die Tastenkombination SHIFT + H aufgerufen wird.

Das Fenster soll folgendermaßen angezeigt werden:

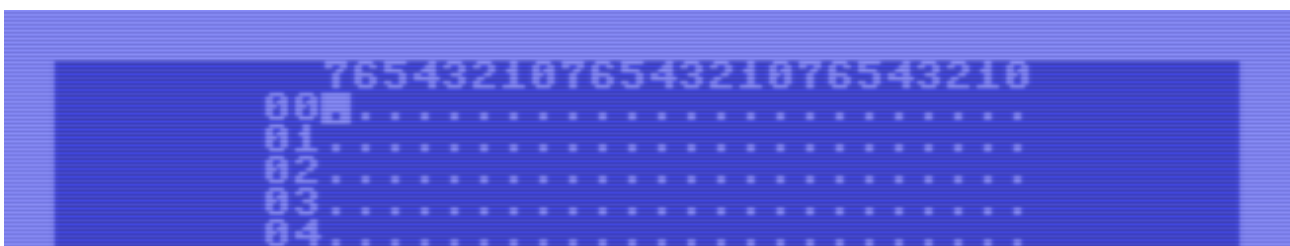


Geschlossen wird das Fenster durch einen beliebigen Tastendruck.

Sehen wir uns den grundsätzlichen Ablauf an.

Bevor das Fenster eingeblendet wird, müssen wir den Inhalt der obersten 6 Zeilen sichern, da nach dem Schließen des Fensters ja wieder der ursprüngliche Inhalt dieser 6 Zeilen sichtbar werden soll.

Ansonsten würde uns folgender Teil des Bildschirmspeichers verloren gehen, da dieser durch die Anzeige des Fensters überschrieben wird.



Um den Inhalt dieser 6 Zeilen zu sichern, müssen wir uns im Datenabschnitt ein Datenfeld definieren, das groß genug ist, um die Zeichen in diesem Bereich aufnehmen zu können.

Wie groß muß dieses Feld sein? Jede Bildschirmzeile besteht aus 40 Zeichen, das ergibt bei 6 Zeilen also 240 Bytes.

Warum ich das Fenster auf 6 Zeilen begrenzt habe, werde ich im Anschluss noch erläutern.

Drückt der Benutzer also die Tastenkombination SHIFT + H, wird zunächst der Inhalt dieser 6 Zeilen in dieses Datenfeld kopiert.

Dann werden die Zeilen gelöscht und das Fenster mit den Hilfsinformation wird eingeblendet. Die Zeilen müssen wir deswegen löschen, da sich der Inhalt des Fensters ansonsten mit dem vorherigen Inhalt der Zeilen vermischen würde.

Das Fenster bleibt solange sichtbar, bis der Benutzer eine beliebige Taste drückt.

Nun muss der ursprüngliche Inhalt der 6 Zeilen wiederhergestellt werden. Dazu kopieren wir die 240 Bytes, die wir zuvor im Datenfeld gesichert haben wieder in den Bildschirmspeicher.

Warum habe ich den Umfang des Fensters auf 6 Zeilen begrenzt? Die Ursache ist in der indizierten Adressierung zu finden.

Die Indizierung findet entweder über das X Register oder das Y Register statt. Da diese Register nur ein Byte fassen können, liegt der maximale Index also bei 255. Wir können mittels der indizierten Adressierung also nur einen Speicherbereich ansprechen, der maximal 256 Bytes umfasst.

Und da das Sichern, Löschen und Wiederherstellen der Zeilen durch indizierte Adressierung stattfindet, sind wir hier auf eine maximale Größe von 256 begrenzt.

Bereits ab 7 Zeilen würden wir den Maximalwert überschreiten, da 7 Zeilen bereits 280 Bytes umfassen.

Natürlich gibt es Möglichkeiten, größere Bereiche des Bildschirms anzusprechen. Man müsste dann allerdings eine Logik umsetzen, die den Bereich in mehrere Teile zu je 256 Bytes, also in Pages, unterteilt und den Zugriff entsprechend über die nummerierten Pages realisieren.

Ich wollte den Sprite-Editor jedoch nicht noch komplizierter machen und deshalb habe ich das Hilfefenster auf 6 Zeilen begrenzt, damit wir in Bezug auf den Index mit einem Byte auskommen.

Beginnen wir mit der Definition des Datenfeldes, welches zur Sicherung der 6 Bildschirmzeilen dient. Ich habe das Datenfeld screenbuffer genannt, da dies relativ gut den Sinn und Zweck des Feldes beschreibt.

```
screenbuffer
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
.byte $00, $00, $00, $00, $00, $00, $00
```

```

.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00
.byte $00,$00,$00,$00,$00,$00,$00

```

Doch mit dem Feld allein ist es nicht getan, wir benötigen noch diverse Unterprogramme, die damit arbeiten.

Folgende drei Unterprogramme müssen wir erstellen:

- **savescrlines:** Sichert die ersten 6 Zeilen des Bildschirms in das Datenfeld scrbuffer
- **clrscrlines:** Löscht die ersten 6 Zeilen des Bildschirms, in dem es sie mit Leerzeichen füllt
- **restorescrlines:** Stellt den vorherigen Inhalt der ersten 6 Zeilen des Bildschirms wieder her

Diese drei Unterprogramme sind im folgenden aufgeführt und sollten aufgrund der Kommentare selbsterklärend sein, sodaß es keine umfassenden Erläuterungen von meiner Seite mehr bedarf.

Unterprogramm savescrlines

```

-----
;
; savescrlines
; sichert den inhalt der ersten 6
; bildschirmzeilen in das datenfeld
; scrbuffer
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,status
;
savescrlines
    ldx #$00
savelineloop
    ; zeichen aus dem
    ; bildschirmspeicher holen
    lda $0400,x
    ; im datenfeld screenbuffer
    ; ablegen
    sta screenbuffer,x
    ; index auf naechstes zeichen
    inx
    ; bereits 240 zeichen gelesen?

```

```

        cpx #$f0
        ; falls nicht, naechstes
        ; zeichen sichern
        bne savelineloop
        ; ansonsten sind wir fertig
        rts

```

Unterprogramm clrscrlines

```

-----
: clrscrlines
: loescht die ersten 6 zeilen des
: bildschirms, diese werden mit
: leerzeichen gefuelllt
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,status
:
clrscrlines
        ; fuellzeichen ist leerzeichen
        lda #$20
        ldx #$00
clrlineolop
        ; leerzeichen schreiben
        sta $0400,x
        ; index auf naechste position
        inx
        ; schon alle 240 positionen
        ; durchlaufen?
        cpx #$f0
        ; falls nicht, weiter zur
        ; naechsten position
        bne clrlineolop
        ; ansonsten sind wir fertig
        rts

```

Unterprogramm restorescrlines

```
-----
: restorescrlines
: stellt die gesicherten ersten 6
: bildschirmzeilen wieder her
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,status
restorescrlines
    ldx #$00
restorelineloop
    ; zeichen aus datenfeld
    ; screenbuffer holen
    lda screenbuffer,x
    ; in den bildschirmspeicher
    ; schreiben
    sta $0400,x
    ; index auf naechstes zeichen
    inx
    ; schon alle 240 zeichen
    ; wiederhergestellt?
    cpx #$f0
    ; falls nicht, naechstes
    ; zeichen wiederherstellen
    bne restorelineloop
    ; ansonsten sind wir fertig
    rts
```

Nun müssen wir die Aufrufe dieser drei Unterprogramme in der richtigen Reihenfolge kombinieren. Dazu schreiben wir ein Unterprogramm showhelp, welches beim Drücken der Tastenkombination SHIFT + H aufgerufen wird.

Das Unterprogramm ist noch nicht ganz fertig, es fehlt noch die Ausgabe der Hilfetexte, aber der grundsätzliche Ablauf aus Sichern, Löschen und Wiederherstellen der Bildschirmzeilen funktioniert bereits.

```
-----
: showhelp
: zeigt ein fenster mit hilfstexten an
:
: parameter:
: keine
:
: rueckgabewerte:
: keine
:
: aendert:
: a,x,y,status
showhelp
```

```

        ; die ersten 6 zeilen des
        ; bildschirms sichern
        jsr savescrlines
        ; zeilen loeschen
        jsr clrscrlines
        ; auf tastendruck warten
waitkeyloop
        jsr $ffe4
        beq waitkeyloop
        ; zeilen wiederherstellen
        jsr restorescrlines
        rts

```

Um die Hilfsfunktion auch über die Tastatur verfügbar zu machen, müssen wir wieder das Unterprogramm keyctrl anpassen und die Abfrage der neuen Tastenkombination integrieren.

```

check_clear
        ; sprite loeschen?
        cmp clearkey
        bne check_help
        jsr clrsprite
        jmp keyloop
check_help
        ; fenster mit hilfetexten
        ; anzeigen?
        cmp helpkey
        bne check_quit
        jsr showhelp
        jmp keyloop

```

Dazu ergänzen wir, wie bereits mehrfach bei den anderen Programmfunktionen durchgeführt, einen entsprechenden Abschnitt namens check_help, in dem auf die Tastenkombination SHIFT + H reagiert wird, indem das zuvor definierte Unterprogramm showhelp aufgerufen wird.

Im Abschnitt check_clear mussten wir auch das Sprungziel nach dem Befehl BNE von check_quit auf check_help abändern, damit die neue Tastenkombination in die Kette der Vergleichsabfragen integriert wird.

Den aktuellen Stand des Programms finden Sie unter dem Namen SAVESCR auf der Diskette. Wenn Sie die Tastenkombination SHIFT + H drücken, werden die ersten 6 Zeilen des Bildschirms gelöscht und nach einem beliebigen Tastendruck ihr ursprünglicher Inhalt wiederhergestellt.

Nun müssen wir nur noch den Rahmen um das Fenster bzw. die Hilfsinformationen darin anzeigen und wir haben unseren Sprite-Editor fertiggestellt!

Dazu definieren wir uns im Datenabschnitt zunächst die Texte, die wir in dem Fenster anzeigen wollen.

```

helpsetreset
    .text "set/reset bit: "
    .null "return/space"

helpsaveload
    .text "save/load: shift + s"
    .null " / shift + l"

helpclear
    .text "clear: shift + clear/"
    .null "home"

helpquit
    .null "quit: shift + q"

```

Für das Befüllen des Hilfefensters habe ich ebenfalls ein eigenes Unterprogramm namens drawhelpwindow geschrieben. Das wäre zwar nicht unbedingt nötig gewesen, aber ich finde, es sorgt für ein wenig mehr Übersichtlichkeit.

```

;-----
; drawhelpwindow
; zeichnet einen rahmen um die ersten
; 6 bildschirmzeilen und gibt darin
; hilfsinformationen aus
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
drawhelpwindow
    ; ecke links oben
    lda #$55
    sta $0400

    ; ecke rechts oben
    lda #$49
    sta $0427

    ; ecke links unten
    lda #$4a
    sta $04c8

    ; ecke rechts unten
    lda #$4b
    sta $04ef

    ; horizontale rahmenlinien
    lda #$40
    ldx #$00

horizloop
    sta $0401,x
    sta $04c9,x
    inx
    cpx #$26
    bne horizloop

    ; vertikale rahmenlinien
    lda #$42

    sta $0428
    sta $044f

```

```

sta $0450
sta $0477
sta $0478
sta $049f
sta $04a0
sta $04c7

; hilfetexte ausgeben

ldx #$01
ldy #$06
lda #<helpsetreset
sta $fd
lda #>helpsetreset
sta $fe
jsr printstr

ldx #$02
ldy #$04
lda #<helpsaveload
sta $fd
lda #>helpsaveload
sta $fe
jsr printstr

ldx #$03
ldy #$05
lda #<helpclear
sta $fd
lda #>helpclear
sta $fe
jsr printstr

ldx #$04
ldy #$0b
lda #<helpquit
sta $fd
lda #>helpquit
sta $fe
jsr printstr

rts

```

Hier werden zuerst die abgerundeten Ecken gezeichnet. Dazu wird einfach der Screencode des jeweiligen Zeichens in den Akkumulator geschrieben und an die entsprechende Adresse im Bildschirmspeicher geschrieben.

Als nächstes sind die horizontalen Linien am oberen und unteren Rand des Rahmens an der Reihe.

Dies wird über eine Schleife realisiert, in der abwechselnd ein Teilstrich am oberen und unteren Rand in den Bildschirmspeicher geschrieben wird.

Die Teile der vertikalen Linien habe ich einzeln in den Bildschirmspeicher geschrieben, da es nicht so viele sind und eine Schleife keinen Mehrwert gebracht hätte.

Somit ist der Rahmen schon mal gezeichnet.

Nun müssen wir nur noch die Hilfetexte in dem Fenster ausgeben.

Dies wird über das bereits bekannte Unterprogramm printstr realisiert, das wir ganz zu Beginn unserer Arbeit erstellt haben.

Als Parameter werden die gewünschte Position des Strings im X Register und Y Register sowie die Stringadresse aufgeteilt auf die Speicherstellen \$FD und \$FE übergeben.

Dieses Schema wenden wir auf alle Strings an, die wir uns vorhin für die Hilfetexte definiert haben.

Nun müssen wir im Unterprogramm showhelp nur noch den Aufruf des Unterprogramms drawhelpwindow einbauen und wir haben auch die Hilfefunktion erfolgreich umgesetzt.

```
; zeilen loeschen
jsr clrscrlines
; hilfsinformationen anzeigen
jsr drawhelpwindow
```

Die finale Fassung des Spriteeditors habe ich unter dem Namen SPRITEEDITOR auf der Diskette zur Verfügung gestellt.

So, nun haben wir es endlich geschafft und unser Projekt Sprite-Editor erfolgreich umgesetzt!

Wie eingangs erwähnt, habe ich ihn so programmiert, dass noch Platz für Verbesserungen bleibt, die Sie als Übung nach Ihren eigenen Vorstellungen umsetzen können wenn sie gerne möchten. Ich hoffe, dass ich Ihnen mit diesem Buch einen erfolgreichen Einstieg in die Assembler-Programmierung auf dem Commodore 64 ermöglichen konnte.

Wenn Sie mit den erworbenen Fähigkeiten nun Ihre eigenen Assembler-Programme erstellen können, dann habe ich mein Ziel erreicht :)

Erinnern Sie sich noch an unser erstes Maschinen-Programm? Wir haben eine Zahl in das X Register geschrieben und sind dann gleich wieder zu Basic zurückgekehrt. Das Programm bestand also aus ganzen 3 Bytes und nun sehen Sie sich an, wo wir heute stehen!

Wir haben ein ausgewachsenes Assembler-Programm geschrieben, das nicht nur einen praktischen Nutzen hat, sondern Ihnen (so hoffe ich) auch einen erfolgreichen Einstieg in diese Art der Programmierung verschafft hat.