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.

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. In diesem Kapitel werde ich Ihnen Schritt für Schritt die einzelnen Codefunktionen näherbringen, aus denen der Sprite-Editor besteht. Zusammengesetzt ergeben diese dann ein ganz ansehnliches Assembler-Programm, welches meiner Meinung nach auch einen sehr guten Abschluss für dieses Buch darstellt.

Zunächst müssen wir uns jedoch 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:

Zeile * 40 + 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 Ausführung des Befehls ASL:

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	0	0	1	0	0	0

Inhalt des Akkumulators nach Ausführung des Befehls ASL:

7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
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.

An dieser Stelle möchte ich Ihnen nun den Befehl ROL (rotate left) vorstellen, da wir ihn dann gleich bei der Umsetzung der 16bit Version des Befehls ASL benötigen werden. 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 Ausführung des Befehls ROL und angenommen, das Carryflag hat den Inhalt 0:

7	6	5	4	3	2	1	0
1	1	0	0	1	0	0	0

Inhalt des Akkumulators nach der Ausführung des Befehls ROL:

7	6	5	4	3	2	1	0
1	0	0	1	0	0	0	0

Auf der rechten Seite ist ein Bit mit dem Inhalt 0 hereingewandert (ersichtlich durch das orange markierte 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 nach der Ausführung folgenden Inhalt:

7	6	5	4	3	2	1	0
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. Daran ändert sich bei diesem Schritt auch nichts, weil das blau markierte Bit ebenfalls mit dem Inhalt 1 ins Carryflag gewandert ist.

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.

Bevor wir nun zur konkreten Umsetzung des Unterprogramms asl16 kommen, noch ein paar Worte zum Thema Dokumentation des geschriebenen Assemblercodes.

Werfen Sie einen Blick auf den Kommentarblock zu Beginn des Unterprogramms, dessen Zeilen mit einem Semikolon eingeleitet werden.

```
;------------------------------------
; asl16
; 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

asl16

asl16_loop
          ; lo byte shiften

          asl $fd

          ; hi byte shiften

          rol $fe

          dex
          bne asl16_loop

          rts
```

Hier ist alles enthalten, was man wissen muss, um dieses Unterprogramm nutzen zu können. Da wäre natürlich als Erstes der Name des Unterprogramms (asl16) gefolgt von Angaben zu den eventuell nötigen Parametern bzw. Rückgabewerten, welche das Unterprogramm als Ergebnisse liefert.

Anmerkung: Mit **lo** meine ich hier das niederwertige und mit **hi** das höherwertige Byte einer 16bit Zahl.

Am Ende habe ich noch eine Angabe darüber gemacht, welche Inhalte in dem Unterprogramm verändert werden. In diesem Fall wird durch das Unterprogramm der Akkumulator, das X Register und das Statusregister verändert.

Dieses Schema werde ich ab jetzt bei jedem Unterprogramm anwenden, damit man auf den ersten Blick erfährt, was das Unterprogramm macht, welche Parameter es entgegennimmt, welche Rückgabewerte es als Ergebnisse liefert und welche Inhalte innerhalb des Unterprogramms verändert werden.

Doch nun zur technischen Umsetzung. 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 ergibt sich folgender Wert:

7	6	5	4	3	2	1	0
1	0	0	0	1	0	0	0

Auf der rechten Seite ist ein Bit mit dem Inhalt 0 hereingewandert und auf der linken Seite ein Bit herausgefallen und ins Carryflag gewandert. 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
0	0	0	1	0	0	1	0

Auf der rechten Seite ist ein Bit mit dem Inhalt 0 hereingewandert und auf der linken Seite ein Bit herausgefallen und ins Carryflag gewandert. 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? Man kann sich natürlich eine umständliche Logik bauen, welche für den Fall, dass das herausgefallene Bit den Inhalt 1 hatte, das höherwertige Byte entsprechend korrigiert. Es gibt jedoch einen einfacheren Weg und hier kommt der Befehl ROL ins Spiel.

Falls bei der Bitverschiebung des niederwertigen Bytes auf der linken Seite ein Bit mit dem Inhalt 1 herausfällt, dann würde das Carryflag den Inhalt 1 bekommen.

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 ergibt sich folgender Wert:

7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	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.

```
asl16
asl16_loop
          ; lo byte shiften
          asl $fd
          ; hi byte shiften
          rol $fe
          dex
          bne asl16_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 schießlich 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. 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

Inhalt des X Registers + 256 * 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 diesesmal 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 calcposaddr 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 calcposaddr 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 calcposaddr

          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 calcposaddr 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.

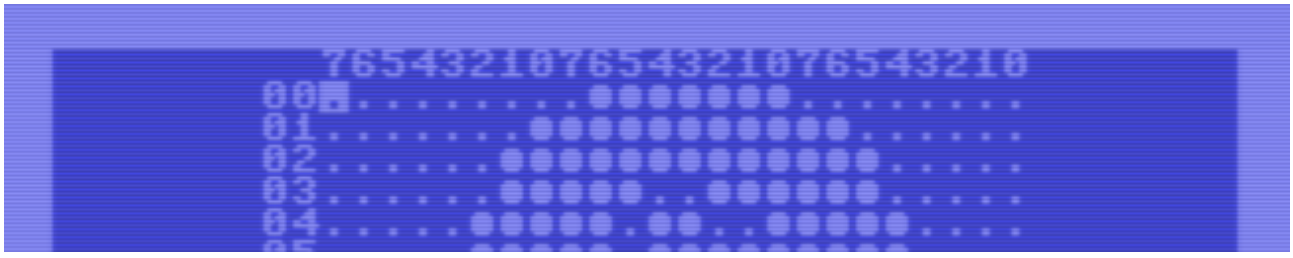
Wenn wir den Sprite-Editor starten, dann ist folgendes Bild zu sehen:

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.

Ein kleiner Punkt bedeutet eine „0“ und ein großer Punkt bedeutet eine „1“.



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

Dieser Cursor sieht zwar aus wie der Cursor, den wir gewohnt sind, er ist jedoch völlig unabhängig von diesem und eigentlich nur ein Zeichen im Bildschirmspeicher. 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 calcposaddr 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.

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 duch 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 Kernal-Funktion 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.

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.

Auf die Bedeutung des Kommas gefolgt von dem Y kommen wir noch zu sprechen.

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 diesesmal 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

Inhalt der Speicherstelle $FB + 256 * 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 diesesmal die Zeichenfolge „#>“ zu sehen ist.

```
          lda #>bitposstr
```

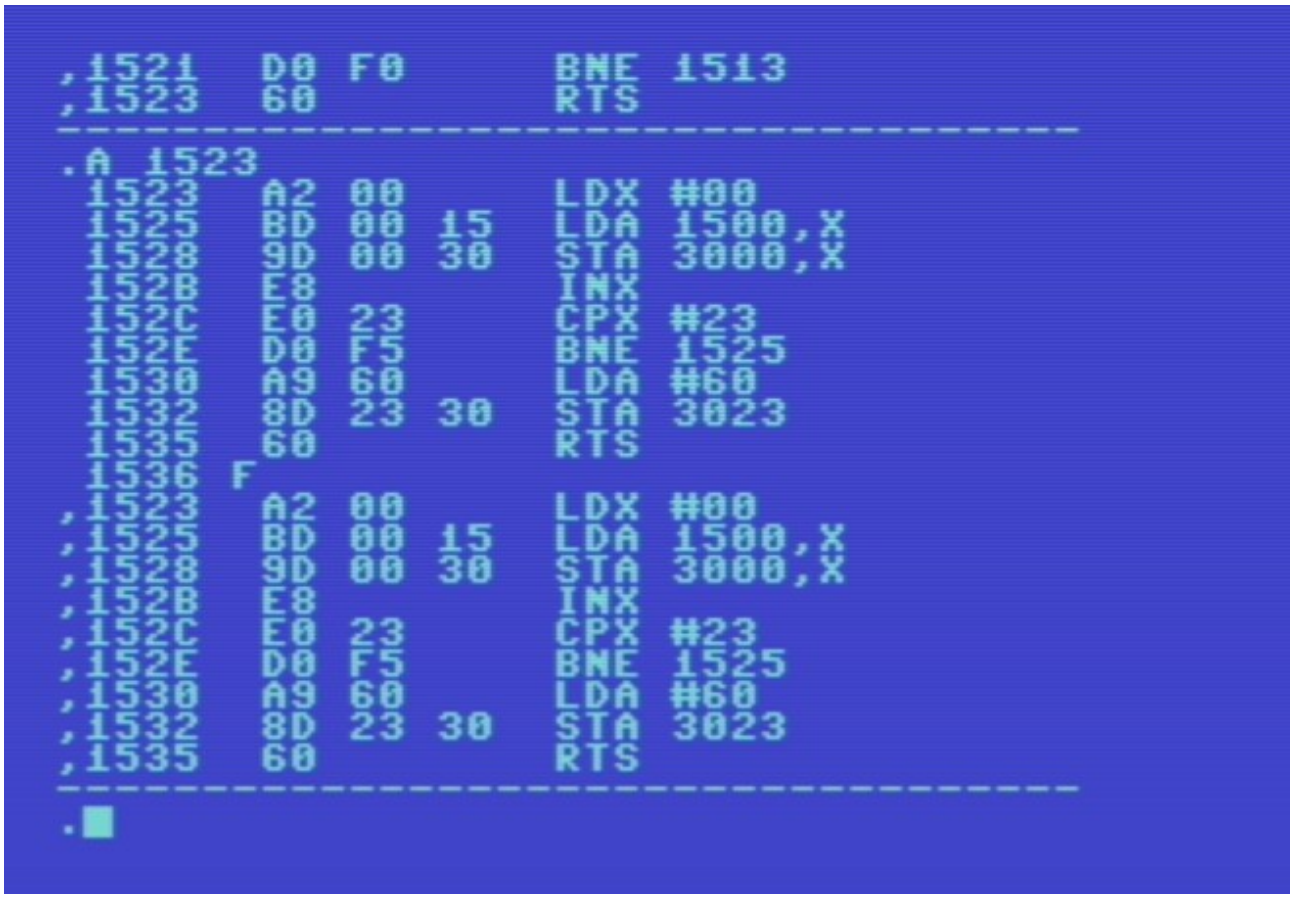
Was hat es damit auf sich?

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:



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 diesesmal 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 Cursorposition 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 Reihennummer 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 ausgegeben, 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 ausgebe.

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

```
scrrow
          .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 scrrow 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 scrrow

          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 scrrow
```

```
            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 scrrow
            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……....

Reihennummer	Index in Datenfeld rownrstr	Nummernstring aus Datenfeld rownrstr	editorrowstr
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 scrrow.

Daher kopieren wir deren Inhalt mit dem Befehl LDX scrrow 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.

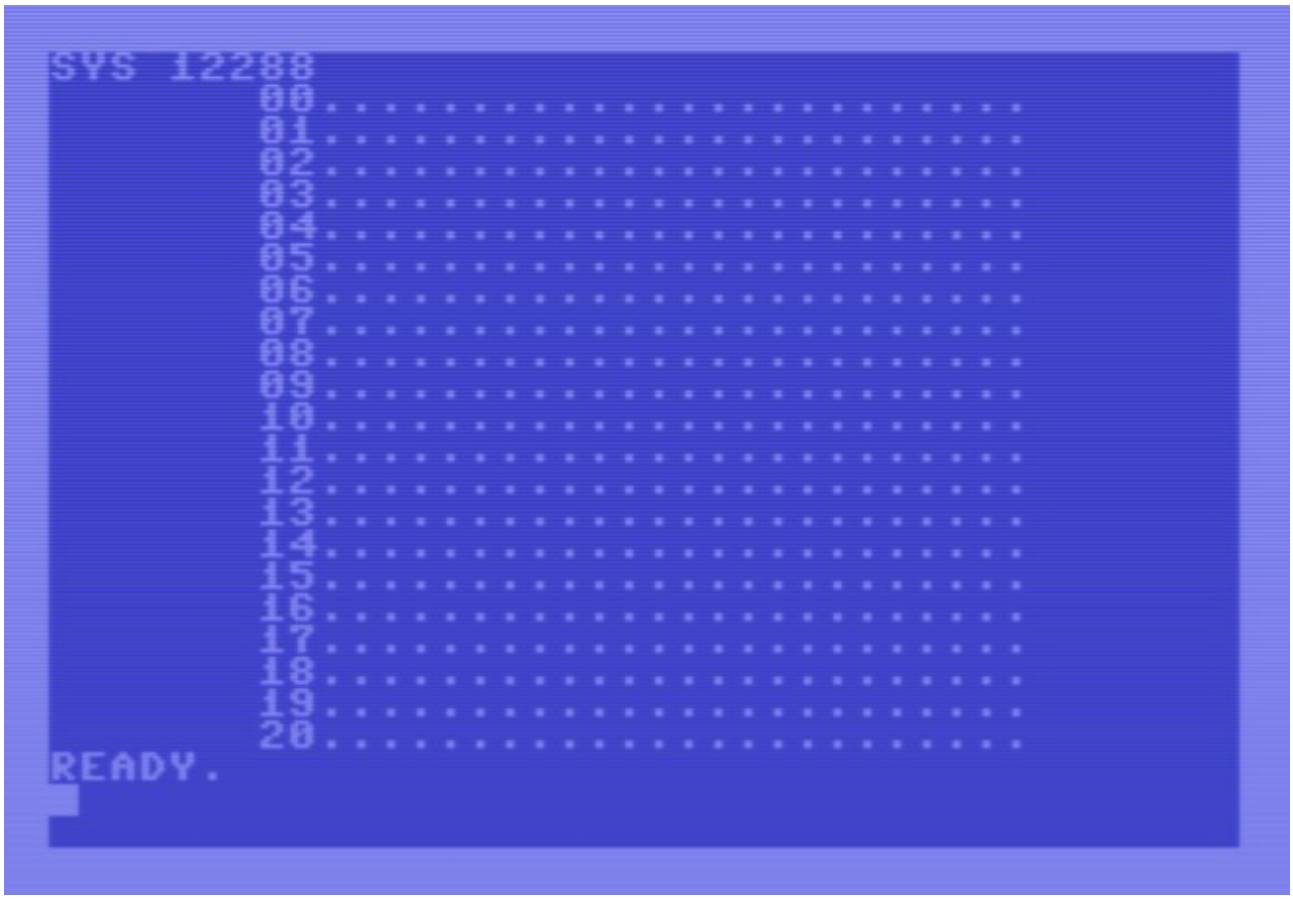
Falls jedoch noch nicht alle Reihen ausgegeben wurden, erhöhen wir die Reihennummer im Akkumulator um 1, um zur nächsten Reihe überzugehen.

Dasselbe machen wir mit dem Inhalt der Variablen scrrow, 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 scrrow

          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 scrrow
            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 scrrow
            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.