

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 nun 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 bauen, die wir im Laufe der Entwicklung des Sprite-Editors immer wieder benötigen werden.

- Wir brauchen eine 16bit Version des Befehls ASL, also die Möglichkeit eine 16bit Zahl bitweise nach links verschieben zu können.
- Die Möglichkeit zwei 16bit Zahlen zu addieren, also eine 16bit Version des Befehls ADC.
- Aus diesen Teilen setzen wir uns ein Unterprogramm zusammen, mit dem wir eine Position am Bildschirm (gegeben durch Zeile / Spalte) in eine Adresse im Bildschirmspeicher umrechnen können.

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.

Beginnen wir mit der 16bit Version des Befehls ASL Wozu brauchen wir diesen?

Nun, 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 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 auf 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 Werte entspricht dann der Multiplikation der Zahl mit 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 ASL Befehls fertiggestellt haben.

Spielen wir das doch mal anhand eines Beispiels durch. Angenommen, wir haben in unserer Positionsangabe die Zeile 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 der Multiplikation von 18 mit 40.

$$\begin{aligned} 18 * 32 &= 576 \\ 18 * 8 &= 144 \\ 576 + 144 &= 720 \end{aligned}$$

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, der Unterschied ist jedoch der, dass auf der rechten Seite immer ein Bit 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:

Inhalt des Akkumulators vor Ausführung des Befehls ROL, 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 Ausführung des Befehls ROL, das Carryflag hat nun den Inhalt 1, denn das grün markierte Bit ist ins Carry Flag gewandert.

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

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 den Inhalt 1 hatte. Dabei bleibt es auch, weil das grün markierte Bit mit dem Inhalt 1 ins Carryflag gewandert ist.

Auf diese Weise 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.

Werfen Sie nur mal einen Blick auf den Kommentarblock am Anfang, auf den eigentlichen Assemblercode komme ich im Detail zurück, sobald ich Ihnen die rechnerische Vorgehensweise erläutert habe.

```
-----
;
; 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
; 10 byte shiften
    asl $fd
; hi byte shiften
    rol $fe
    dex
    bne asl16_loop
    rts
```

Wie Sie hier sehen können, habe ich vor Beginn des eigentlichen Unterprogramms einen Kommentarblock mit hilfreichen Informationen gestellt. 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 Parametern.

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

Da das Unterprogramm eine 16bit Zahl um eine bestimmte Anzahl an Stellen nach links schieben soll, müssen wir dem Unterprogramm diese 16bit Zahl irgendwo aufgeteilt auf das niederwertige

Byte (lo Byte) und das höherwertige Byte (hi Byte) zur Verfügung stellen. In diesem Fall habe ich mich für die Speicherstellen \$FD und \$FE entschieden, da die Speicherstellen \$FB bis \$FE zur freien Verfügung stehen.

Die Anzahl der Stellen um die wir die Zahl nach links schieben wollen, übergeben wir im X Register.

Das 16bit Ergebnis soll dann aufgeteilt auf die Speicherstellen \$FD und \$FE zurückgeliefert werden.

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 liefert und welche Inhalte innerhalb des Unterprogramms verändert werden.

Doch nun zur technischen Umsetzung. Angenommen wir wollen die 16bit Zahl 2500 um eine Bitposition nach links schieben. 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.

Nun wollen wir diese 16bit Zahl um eine Bitposition nach links schieben. Doch wie machen wir das? 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 wandert dieses ja in das Carryflag.

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.

```
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.

Das Unterprogramm bietet jedoch 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 angekommen 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 oder 128 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 jedoch 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 in das niederwertige 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 in das höherwertige 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: 10/hi in $fb/$fc  
: zahl2: 10/hi im x/y register  
:  
: rueckgabewerte:  
: summe: 10/hi in $fb/$fc  
:  
: aendert:  
: a,status  
:  
adc16  
    ; 10 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 von Zeile / Spalte 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 der Zeile 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
:
:   ld y # $00
:   sty $fe
:
:   ; wir wollen mit 32, also
:   ; mit 2 hoch 5 multiplizieren,
:   ; daher 5 stellen als
:   ; parameter im x register
:   ; uebergeben
:
:   ld x # $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 Wert für die Zeile 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, in diesem Fall 5, denn wir wollen den Zeilenwert ja mit 32 multiplizieren, was einer Verschiebung von 5 Bitpositionen nach links entspricht.

Nach dem Aufruf von asl16 liegt 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.

Diese beiden Werte müssen wir uns irgendwo merken, weil wir die Speicherstellen \$FD und \$FE für die nächste Multiplikation brauchen. In diesem Fall merken wir sie uns 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.

Der Zeilenwert liegt an oberster Stelle des Stacks, d.h. wir können den Wert 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

```
; 10 (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, müssen wir das X Register mit dem Wert 3 laden.

Nach dem Aufruf von asl16 ist das Ergebnis wieder in den Speicherstellen \$FD und \$FE vorhanden.

Nun müssen wir diese beiden Werte 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 eine der beiden Zahlen die addiert werden sollen, bereits wie von adc16 erwartet, in den Speicherstellen \$FB und \$FC vorliegt, brauchen wir sie nicht extra dorthin transportieren, sondern wir müssen nur das niederwertige Byte der zweiten Zahl im X Register und das höherwertige Byte der zweiten Zahl im Y Register bereitstellen. Letzteres erfolgt über die Befehle LDX \$FD und LDY \$FE.

Nach dem Aufruf von adc16 steht die Summe aus Zeilenwert * 32 und Zeilenwert * 8 aufgeteilt auf die Speicherstellen \$FB und \$FC zur Verfügung.

Nun müssen wir zu diesem Wert den Spaltenwert addieren. Dieser liegt griffbereit auf der obersten Stelle des Stacks und 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 liegt 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) liegt noch immer in den Speicherstellen \$FB und \$FC.

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 diesmal unserem Endergebnis entspricht, also der gewünschten Adresse im Bildschirmspeicher.

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:

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

Passt also!