

Inhaltsverzeichnis

| | |
|---|-----|
| Vorwort..... | 3 |
| 1 Programmieren in Maschinensprache..... | 4 |
| 1.1 Die CPU-Register..... | 5 |
| 1.2 Der Arbeitsspeicher..... | 5 |
| 1.3 Erste Programme in Maschinensprache..... | 7 |
| 1.3.1 Programm 1: Laden des X Registers..... | 7 |
| 1.3.2 Programm 2: Ausgabe eines Zeichens auf dem Bildschirm..... | 9 |
| 1.3.3 Erstellung von BASIC-Loadern..... | 12 |
| 1.3.4 Laden des X Registers aus unterschiedlichen Quellen..... | 13 |
| 1.3.5 Programm 3: Einfache Rechenoperationen (Version 1)..... | 15 |
| 1.3.6 Programm 4: Einfache Rechenoperationen (Version 2)..... | 17 |
| 2 Programmieren in Assembler..... | 22 |
| 2.1 Warum Assembler?..... | 22 |
| 2.2 Einführung in die Arbeit mit SMON..... | 29 |
| 2.2.1 Was ist SMON?..... | 29 |
| 2.2.2 Laden und Starten von SMON..... | 30 |
| 2.2.3 Eingabe von Assembler-Befehlen..... | 31 |
| 2.2.4 Starten eines Assembler-Programms..... | 38 |
| 2.2.5 Anzeigen der aktuellen Register-Inhalte..... | 39 |
| 2.2.6 Anzeigen des Inhalts von Speicherbereichen..... | 39 |
| 2.2.7 Disassemblieren – aus Maschinensprache wird wieder Assembler..... | 42 |
| 2.2.8 Speichern eines Assembler-Programms auf Diskette..... | 46 |
| 2.2.9 Laden eines Assembler-Programms von Diskette..... | 47 |
| 2.2.10 Automatisches Erstellen von DATA-Zeilen für einen BASIC-Loader..... | 48 |
| 3 Von Adressierungsarten, Schleifen und Programmen, die sich selbst teleportieren..... | 56 |
| 3.1 Adressierungsarten..... | 57 |
| 3.1.1 Unmittelbare Adressierung..... | 57 |
| 3.1.2 Absolute Adressierung..... | 57 |
| 3.1.3 Zeropage Adressierung..... | 58 |
| 3.1.4 Implizite Adressierung..... | 58 |
| 3.1.5 X-indizierte absolute Adressierung..... | 58 |
| 3.1.6 Y-indizierte absolute Adressierung..... | 59 |
| 3.1.7 X-indizierte Zeropage Adressierung und Y-indizierte Zeropage Adressierung..... | 59 |
| 3.2 Beispiel-Programme..... | 59 |
| 4 Der Stapspeicher und Unterprogramme..... | 84 |
| 4.1 Was ist der Sinn und Zweck des Stacks?..... | 84 |
| 4.2 Befehle zum Zugriff auf den Stack..... | 84 |
| 4.3 Beispiel-Programme..... | 87 |
| 5 Unterprogramme..... | 95 |
| 5.1 Übergabe von Parametern an Unterprogramme..... | 98 |
| 5.1.1 Übergabe der Parameter in CPU Registern..... | 98 |
| 5.1.2 Übergabe der Parameter in ausgesuchten Stellen in der Zeropage..... | 105 |
| 5.1.3 Übergabe der Parameter in einem eigenen Datenbereich an passender Stelle im Hauptspeicher..... | 107 |
| 5.1.4 Übergabe der Parameter auf dem Stack..... | 110 |
| 5.2 Beispiel-Programm..... | 111 |
| 6 Zahlensysteme..... | 121 |
| 6.1 Dezimalsystem, Binärsystem, Hexadezimalsystem..... | 121 |

| | |
|--|------------|
| 6.2 Umrechnen zwischen Zahlensystemen..... | 124 |
| 6.3 Addition von Binärzahlen..... | 131 |
| 6.4 Die Darstellung von negativen Zahlen..... | 138 |
| 7 Logische Verknüpfungen..... | 141 |
| 7.1 UND-Verknüpfung (AND) und ODER-Verknüpfung (OR)..... | 141 |
| 7.2 Bitweises Verknüpfen von Bytes..... | 161 |
| 7.2.1 ODER - Verknüpfung (OR)..... | 161 |
| 7.2.2 UND - Verknüpfung (AND)..... | 162 |
| 7.2.3 EXKLUSIV ODER – Verknüpfung (XOR)..... | 163 |
| 7.2.4 Setzen von Bits..... | 164 |
| 7.2.5 Zurücksetzen von Bits..... | 173 |
| 8 Einführung in die Arbeit mit Turbo Macro Pro..... | 183 |
| 8.1 Eingabe, Assemblierung und Starten eines Assembler-Programms..... | 184 |
| 8.2 Speichern eines Assembler-Programms auf Diskette..... | 190 |
| 8.3 Laden eines Assembler-Programms von Diskette..... | 191 |
| 8.4 Überschreiben eines bereits bestehenden Assembler-Programms..... | 193 |
| 8.5 Erstellen eines eigenständigen Programms aus einem Assembler-Programm..... | 194 |
| 8.6 Löschen von einzelnen Zeilen..... | 197 |
| 8.7 Kommentare..... | 198 |
| 8.8 Einfügen von Trennzeilen..... | 199 |
| 8.9 Blockoperationen..... | 199 |
| 8.10 Schnell-Navigation innerhalb des Assembler-Programms..... | 203 |
| 8.11 Einen Kaltstart des Turbo Macro Pro durchführen..... | 203 |
| 9 Sprites..... | 204 |
| 9.1 Was sind Sprites?..... | 204 |
| 9.2 Einfarbige Sprites..... | 204 |
| 9.3 Erstellung von Sprites..... | 206 |
| 9.4 Auswahl des Speicherblocks für die Spritedaten..... | 208 |
| 9.5 Typ des Sprites festlegen (einfarbig oder mehrfarbig)..... | 210 |
| 9.6 Farbe des Sprites festlegen..... | 211 |
| 9.7 Festlegen der Spriteposition..... | 211 |
| 9.8 Festlegen der Sprite-Priorität in Bezug auf den Hintergrund..... | 212 |
| 9.9 Aktivierung von Sprites..... | 212 |
| 9.10 Horizontale Vergrößerung von Sprites..... | 216 |
| 9.11 Vertikale Vergrößerung von Sprites..... | 217 |
| 9.12 Arbeiten mit mehreren Sprites..... | 217 |
| 9.13 Prioritäten von Sprites..... | 223 |
| 9.14 Beispiel-Programm (einfarbige Sprites)..... | 224 |
| 9.15 Mehrfarbige Sprites..... | 239 |
| 9.16 Beispiel-Programm (mehrfarbige Sprites)..... | 242 |
| 10 Ein Sprite-Editor als Abschlussprojekt..... | 254 |

Vorwort

Willkommen zu meiner Einführung in die Maschinensprache- und Assembler-Programmierung auf dem Commodore 64!

Oft werden die Begriffe Maschinensprache und Assembler gleichwertig verwendet, obwohl es sich nicht exakt um dasselbe handelt. Sie werden die Unterschiede jedoch schon sehr bald kennenlernen. Als ich Mitte der 80er Jahre zu Weihnachten einen Commodore 16 geschenkt bekam, dauerte es nur wenige Tage, bis ich damit begann, die Beispielprogramme aus dem Handbuch abzutippen und auszuprobieren. Durch das wirklich gute BASIC des Commodore 16 machte das Programmieren richtig Spass und schon bald entwickelte es sich zu einer meiner Lieblingsbeschäftigungen.

Ich tippte auch fleißig Programm listings aus Zeitschriften ab und stieß dabei neben den reinen BASIC-Programmen natürlich unweigerlich auf diese seltsam anmutenden Programme, welche zum größten Teil aus irgendwelchen Zahlen bestanden, die in DATA-Zeilen untergebracht waren.

Oft traf ich auch auf Programme in Form von abgedruckten Listings in Assemblersprache. Nun war die Verwirrung natürlich komplett. Was ist nun Maschinensprache und was hat es mit Assembler auf sich? Ich konnte weder mit dem einen noch mit dem anderen etwas anfangen.

Es wurden zwar in diversen Zeitschriften Kurse zum Thema Maschinensprache und Assembler angeboten, aber selbst mit deren Hilfe fand ich keinen nennenswerten Zugang zu dieser Art der Programmierung.

Kurz darauf wechselte ich vom Commodore 16 auf meinen ersten PC und die Welt der Homecomputer geriet dadurch relativ schnell in Vergessenheit.

Viele, viele Jahre später kamen jedoch nostalgische Gefühle auf und ich ersteigte auf Ebay einen Commodore 64. Zunächst fristete er aus Zeitgründen ein trauriges Dasein in der Vitrine, doch dann kam langsam wieder das Interesse an der Programmierung in Maschinensprache / Assembler auf.

Durch die vielen Möglichkeiten, die sich mittlerweile durch das Internet boten, konnte ich einen ersten Schritt in dieser Richtung hinter mich bringen, der mir damals in meiner Jugend gefehlt hatte. Es war das berühmte Aha-Erlebnis und da ich in der Zwischenzeit auf dem PC Erfahrungen in der Assembler-Programmierung sammeln konnte, fand ich auf einmal den diesbezüglichen Zugang auch auf dem Commodore 64.

Ab diesem Zeitpunkt war ich nicht mehr zu bremsen und das Ergebnis sehen Sie nun vor sich :)

Ich hoffe, dass ich Ihnen mit diesem Buch den Zugang zur Programmierung in Maschinensprache / Assembler auf dem Commodore 64 erleichtern kann und Ihnen das Programmieren ebenso viel Spaß machen wird wie mir :)

Doch nun genug der vielen Worte – starten wir den Commodore 64 und machen uns an die Arbeit!

1 Programmieren in Maschinensprache

Wenn wir unseren C64 einschalten, können wir sofort in BASIC programmieren.



Möglich wird das durch den BASIC-Interpreter, der unser BASIC-Programm Befehl für Befehl in Maschinensprache übersetzt. Die Maschinensprache ist die einzige Sprache, welche die CPU (Central Processing Unit) des C64 (6510 von MOS Technology) „versteht“.

Die CPU ist jener Chip im C64, welcher für die Ausführung von Programmen zuständig ist, sie ist also quasi das „Gehirn“ des C64.

Sämtliche Programme müssen letztendlich irgendwie in Maschinensprache übersetzt werden, damit die CPU etwas damit anfangen kann.

Aber was steckt genau hinter dem Begriff Maschinensprache und wie kann man in dieser Sprache programmieren?

Dazu muss ich etwas ausholen und zunächst einige Begriffe bzw. Sachverhalte erklären.

Um die Erklärung zu vereinfachen, bauen wir unseren C64 gedanklich auseinander und lassen mal alles bis auf die CPU und den Arbeitsspeicher weg. Im Arbeitsspeicher liegen unsere Programme und Daten, welche die CPU verarbeitet.

1.1 Die CPU-Register

Die CPU im C64 verfügt über 6 sogenannte Register, das sind Speicherzellen direkt auf dem Chip, in denen man jeweils genau 1 Byte (8 Bit), also eine Zahl zwischen 0 und 255, ablegen kann. Man nennt diesen Vorgang auch oft Laden des Registers.

Die Namen der Register lauten:

A (auch Akkumulator oder einfach nur Akku genannt)

X

Y

SP (Stackpointer)

SR (Status Register)

PC (Program Counter)

Machen Sie sich im Moment noch keine tieferen Gedanken über die Register, merken Sie sich für's Erste nur, dass man darin Zahlen zwischen 0 und 255 abspeichern kann. Eine Ausnahme bildet das Program Counter Register, dieses kann Werte zwischen 0 und 65535 aufnehmen (2 Bytes oder 16 Bit)

Direkt verändern kann man als Programmierer nur die Register A, X und Y sowie in eingeschränkter Weise das Status Register. Indirekt über den Umweg über das X Register ist auch eine Änderung am Inhalt des SP-Registers möglich.

Wir werden auf die Bedeutung und Verwendungsmöglichkeiten der Register noch genau zu sprechen kommen. Für den Moment müssen Sie sich, wie gesagt, nur merken, dass es Speicherstellen sind, die sich direkt auf der CPU befinden.

1.2 Der Arbeitsspeicher

Kommen wir nun zum Arbeitsspeicher.

Der Arbeitsspeicher des C64 besteht aus 65536 Speicherzellen, welche von 0 bis 65535 durchnummeriert sind.

Sie können sich den Speicher des C64 wie eine lange Straße mit 65536 Häusern vorstellen, wobei das erste Haus die Hausnummer 0 und das letzte Haus die Hausnummer 65535 hat.

In jeder dieser Speicherzellen kann man eine Zahl zwischen 0 und 255, also ein Byte, ablegen. Im Arbeitsspeicher werden sowohl Programme, als auch die Daten, mit denen diese Programme arbeiten, abgelegt.

CPU und Arbeitsspeicher sind über Leitungen miteinander verbunden, über die der Informations-Austausch stattfindet, denn die CPU muss sich ja die einzelnen Befehle eines Programms aus dem Arbeitsspeicher holen, um diese der Reihe nach ausführen zu können.

Die Daten, mit denen das Programm arbeitet, werden ebenfalls über diese Leitungen aus dem Arbeitsspeicher gelesen und die CPU benutzt diese Leitungen im umgekehrten Weg auch, um Werte in den Arbeitsspeicher zu schreiben.

In Sachen Geschwindigkeit besteht ein großer Unterschied zwischen dem Zugriff auf die CPU-Register und dem Zugriff auf die Speicherzellen im Arbeitsspeicher.

Da sich alle Register direkt auf dem CPU-Chip befinden, kann die CPU auf die Werte, welche in diesen Registern gespeichert sind, sehr schnell zugreifen. Braucht die CPU hingegen Daten aus dem Arbeitsspeicher, dauert das natürlich länger, weil sie ja den Umweg über die Verbindungsleitungen nehmen muss um an die Daten zu kommen.

Aber was versteht man nun unter Maschinensprache?

Jede CPU verfügt über eine bestimmte Anzahl an einfachen Befehlen, z.B.

- Addiere zwei Zahlen
- Lade ein bestimmtes Register mit einem Wert
- Kopiere den Inhalt einer bestimmten Stelle im Arbeitsspeicher in ein bestimmtes Register
- Schreibe den Wert eines bestimmten Registers an eine bestimmte Stelle im Arbeitsspeicher

um nur einige zu nennen.

Die Anzahl und Art der Befehle ist von CPU zu CPU unterschiedlich, die CPU im C64 hat einen anderen Befehlssatz als beispielsweise eine Zilog Z80 oder Intel 8088 CPU.

Deswegen gibt es auch nicht "die" Maschinensprache, sondern viele, so viele wie es CPU-Typen gibt.

Es gibt übrigens auch noch andere Computer, deren CPU die gleiche Maschinensprache „spricht“ wie der C64. Daher wird Ihnen das erworbene Wissen auch auf diesen Computern von Nutzen sein. Natürlich ist jeder Computer anders, aber das grundlegende Prinzip der Programmausführung ist immer gleich.

Jedem Befehl ist eine bestimmte Zahl zwischen 0 und 255 zugeordnet. Welchem Befehl welche Zahl zugeordnet ist, kann man in Listen nachschlagen.

Sie müssen sich diese Zahlen nicht merken, denn erstens kann man diese Zahlencodes auf vielen Seiten im Internet in Erfahrung bringen, wenn man sie wirklich brauchen sollte und zweitens sind sie sowieso nur dann von Bedeutung, wenn man, so wie wir hier in den ersten Beispielen, wirklich in Maschinensprache programmiert.

Sobald wir später dann zur Assembler-Programmierung kommen, sind diese BefehlsCodes sowieso nicht mehr so wichtig, weil die Vermeidung der Konfrontation mit diesen BefehlsCodes eben genau einer der Vorteile der Assembler-Sprache gegenüber der Maschinensprache ist.

1.3 Erste Programme in Maschinensprache

1.3.1 Programm 1: Laden des X Registers

Bevor wir beginnen, beachten Sie bitte folgendes.

Wir werden es in diesem Buch naturgemäß mit vielen Zahlen zu tun haben. Ein und dieselbe Zahl kann man basierend auf unterschiedlichen Zahlensystemen darstellen. Wir werden es hauptsächlich mit drei Zahlensystemen zu tun haben, dem Dezimalsystem, dem Hexadezimalsystem und dem Binärsystem.

Falls Sie mit diesen Zahlensystemen eventuell noch nicht so vertraut sind, finden Sie im Kapitel 6 eine Einführung in dieses Thema.

Damit Ihnen klar ist, welches Zahlensystem ich bei der Angabe von Zahlen gerade meine, werde ich Hexadezimalzahlen das Zeichen \$, Binärzahlen das Zeichen % und Dezimalzahlen kein zusätzliches Zeichen voranstellen.

Nehmen wir als Beispiel die dezimale Zahl 200, dann würde meine Darstellung folgendermaßen aussehen:

| Dezimalsystem | Hexadezimalsystem | Binärsystem |
|---------------|-------------------|-------------|
| 200 | \$C8 | %11001000 |

Doch nun kommen wir zu unserem ersten Programm in Maschinensprache.

Fangen wir ganz einfach an. Unser Programm soll nichts weiter tun, als ein Register, nehmen wir beispielsweise das X Register, mit einem bestimmten Wert zu laden.

Der Befehl, mit dem man das X Register mit einem Wert laden kann, hat den Befehlscode 162. Das entspricht \$A2 in hexadezimaler bzw. %10100010 in binärer Schreibweise.

Will man nun der CPU diesen Befehl erteilen, muss man den entsprechenden Befehlscode 162 in eine Stelle des Arbeitsspeichers schreiben. Welche Stelle man dafür verwendet, ist momentan noch nicht so wichtig, aber nehmen wir mal an, wir lassen für die folgenden Betrachtungen unser Programm beispielsweise an der Speicheradresse 5376 beginnen.

Der erste Schritt würde also darin bestehen, den Befehlscode 162 an die Speicheradresse 5376 zu schreiben.

Und das ist überhaupt kein Problem, denn dies können wir aus BASIC heraus mit dem Befehl
POKE 5376,162

durchführen.

Nun müssen wir nur noch angeben, welche Zahl wir denn in das X Register schreiben wollen. Diesen Wert schreibt man dann in die darauffolgende Speicherstelle, also an die Adresse 5377.

Angenommen, man will also den dezimalen Wert 200 in das X Register schreiben. Dann müsste man den Wert 200 in die Speicherstelle mit der Adresse 5377 eintragen.

Analog zu vorhin führen wir dies mit dem Befehl

POKE 5377,200

durch.

An den Speicheradressen 5376 bzw. 5377 stehen nun die Werte 162 und 200 oder in hexadezimaler Schreibweise \$A2 und \$C8 bzw. %10100010 und %11001000 in binärer Schreibweise.

An die Adresse 5378 würden wir dann den Befehlscode des nächsten Befehls unseres Programms schreiben und in den nachfolgenden Speicherstellen würden eventuelle Parameter folgen, die dieser Befehl benötigt.

Auf diese Parameter folgt dann der Befehlscode des nächsten Befehls und dies setzt sich solange fort, bis das Ende des Programms erreicht wird.

Es gibt auch Befehle, die keine Parameter benötigen, dann würde auf den Befehlscode gleich der Befehlscode des nächsten Befehls folgen.

Zur Vervollständigung unseres Programms müssen wir abschließend noch einen Befehl in den Speicher schreiben, der unser Programm beendet und uns wieder zu BASIC zurückbringt.

Die Beendigung des Programms wird meistens durch den Befehlscode 96 erfolgen, oder aber auch durch den Befehlscode 0.

Die unterschiedlichen Auswirkungen dieser beiden Befehle werden wir noch kennenlernen.

Der abschließende Befehl zum Beenden des Programms ist wichtig, da die CPU ansonsten ihre Arbeit einfach mit jenem Byte fortsetzt, das auf das letzte Byte unseres Programms (200) folgt.

Und da nicht vorhersehbar ist, welche Bytes nach unserem Programm im Speicher folgen, ist das Verhalten des C64 ebenso unvorhersehbar.

Die CPU kann schließlich nicht „wissen“, wo unser Programm im Speicher aufhört und daher müssen wir das Ende unseres Programms explizit durch diesen Befehl festlegen.

Führen wir also abschließend noch den Befehl

POKE 5378,96

aus, um den Befehl zum Beenden unseres Maschinenprogramms in den Speicher zu schreiben.

Gratulation an dieser Stelle, sie haben nun Ihr erstes Maschinenprogramm in den Speicher geschrieben! Es besteht zwar nur aus einem Befehl, der nichts anderes macht, als den Wert 200 in das X Register zu schreiben und einem zweiten Befehl, der das Programm beendet, aber immerhin!

Unser Programm besteht aus der Zahlenfolge 162 200 96, beansprucht also 3 Bytes im Arbeitsspeicher.

Soweit so gut, doch allein die Tatsache, dass unser Programm im Speicher steht, bewirkt noch nichts.

Wir müssen das Programm, das wir soeben in den Speicher geschrieben haben, natürlich auch ausführen und BASIC bietet uns mit dem Befehl SYS genau diese Möglichkeit.

Unser Programm beginnt an der Adresse 5376 und daher müssen wir zum Starten des Programms SYS 5376 eingeben.

Umgehend meldet sich der C64 wieder mit READY, ohne dass etwas passiert zu sein scheint. Doch unser Programm wurde ausgeführt, auch wenn es nur das X Register mit dem Wert 200 geladen hat und dann sofort wieder zu BASIC zurückgekehrt ist.

Nun wollen wir prüfen, ob alles gut gegangen ist und nach Ausführung des Programms tatsächlich der Wert 200 im X Register steht.

Doch wie können wir von BASIC aus die Inhalte der CPU-Register auslesen? Es gibt ja keinen entsprechenden BASIC-Befehl, mit dem man beispielsweise den Inhalt des Akkumulators oder der anderen Register auslesen kann.

Bevor ich Ihnen die Lösung dieses Problems präsentiere, möchte ich Ihnen etwas mehr über die Arbeitsweise des Befehls SYS erzählen.

Es gibt vier Speicherstellen, welche in Zusammenhang mit dem Befehl SYS zum Befüllen und Auslesen des Akkumulators, des X Registers, des Y Registers und des Statusregisters genutzt werden können.

Die nachfolgende Tabelle zeigt, welche Speicherstelle welchem Register zugeordnet ist.

| Speicherstelle | Register |
|----------------|----------------|
| 780 | Akkumulator |
| 781 | X Register |
| 782 | Y Register |
| 783 | Statusregister |

Bevor der Befehl SYS das Maschinenprogramm an der angegeben Speicherstelle startet, liest er die Inhalte dieser vier Speicherstellen und befüllt mit diesen Werten die vier Register.

Der Inhalt der Speicherstelle 780 kommt in den Akkumulator, der Inhalt der Speicherstelle 781 in das X Register, der Inhalt der Speicherstelle 782 in das Y Register und der Inhalt der Speicherstelle 783 in das Statusregister.

Dann wird das Maschinenprogramm gestartet und es findet in den vier Registern genau jene Werte vor, die man zuvor per POKE-Befehl in die entsprechenden Speicherstellen geschrieben hat und kann sie nach Belieben auslesen und weiterverarbeiten.

Dieses Spiel funktioniert auch in umgekehrter Richtung.

Wird ein Maschinenprogramm durch den Befehl mit dem Befehlscode 96 beendet, so werden vor der Rückkehr zu BASIC noch die aktuellen Inhalte der vier genannten Register in die zugeordneten Speicherstellen geschrieben.

Wieder im BASIC angekommen, kann man diese Registerinhalte dann per PEEK-Befehl auslesen.

Diese Möglichkeit werden wir nun nutzen, um zu prüfen, ob nach der Ausführung unseres Programms auch wirklich der Wert 200 im X Register steht. Da das X Register der Speicherstelle 781 zugeordnet ist, können wir durch den Befehl PRINT PEEK (781) den Inhalt des X Registers erfahren, den es bei der Beendigung des Maschinenprogramms inne hatte.

Und wie auf folgendem Screenshot zu sehen ist, liefert der Befehl erwartungsgemäß als Ergebnis den Wert 200.



1.3.2 Programm 2: Ausgabe eines Zeichens auf dem Bildschirm

Als nächstes wollen wir ein Programm schreiben, das ein erstes sichtbares Ergebnis bewirkt.

Sie werden im wahrsten Sinne des Wortes auf den ersten Blick sehen, ob unser Programm fehlerfrei ausgeführt wurde oder nicht.

Wir wollen mit dem Programm folgende Schritte ausführen:

- Das X Register soll mit dem Wert 4 geladen werden
- Der Inhalt des X Registers (nun also 4) soll an die Adresse 1024 (hexadezimal \$0400) im Arbeitsspeicher geschrieben werden

Warum gerade die Adresse 1024?

Der Grund ist folgender:

Der Bildschirm des C64 enthält 25 Zeilen zu je 40 Zeichen, das macht also insgesamt 1000 Zeichen.

Die Häuser von Hausnummer 1024 bis 2023 im Arbeitsspeicher enthalten die Zeichencodes der Zeichen, welche gerade am Bildschirm zu sehen sind.

Für die Zeichencodes gibt es ebenfalls eine Liste, in der für jedes Zeichen der entsprechende Zeichencode zu finden ist.

Die erste Hausnummer 1024 beinhaltet den Zeichencode des Zeichens in der linken oberen Ecke und die letzte Hausnummer 2023 beinhaltet den Zeichencode des Zeichens in der rechten unteren Ecke.

Der Bereich von Hausnummer 1024 bis Hausnummer 2023 wird daher auch Bildschirmspeicher (oder auch Videospeicher) des C64 genannt.

Wenn wir nun das X Register mit dem Wert 4 laden und dessen Inhalt anschließend an die Speicheradresse 1024 schreiben, dann steht an dieser Adresse der Zeichencode 4, welcher für den Buchstaben D steht. Das bedeutet, dass nach Ausführung unseres Programms in der linken oberen Ecke ein D zu sehen sein wird.

Wie wir das X Register mit einem Wert laden, wissen wir bereits vom vorherigen Programm. Der Befehlscode lautete 162 und da wir das X Register mit dem Wert 4 laden wollen, müssen wir nachfolgend eine 4 in den Speicher schreiben.

Wir lassen unser Programm wieder an der Adresse 5376 beginnen. Als erstes schreiben wir den Befehlscode zum Laden des X Registers in den Speicher:

POKE 5376,162

Und in die darauffolgende Speicherstelle schreiben wir den Wert 4.

POKE 5377,4

Aber wie kriegen wir den Wert im X Register an die Speicherstelle mit der Nummer 1024?

Dazu brauchen wir den Maschinenbefehl mit dem Code 142, den wir mit dem Befehl

POKE 5378,142

in den Speicher schreiben.

Darauffolgend müssen wir angeben, an welche Speicherstelle wir den Wert aus dem X Register schreiben wollen.

In diesem Fall also die Speicherstelle mit der Nummer 1024.

Hierbei ist folgendes zu beachten:

Der Wert 1024 passt nicht mehr in den Wertebereich eines Bytes, denn dieser umfasst ja nur Werte von 0 bis 255. Um diesen Wert also im Speicher abzulegen, müssen wir ihn in zwei Bytes zerlegen, in das sogenannte niederwertige Byte und das sogenannte höherwertige Byte.

Insgesamt gesehen haben wir es also mit einem 16 Bit – Wert zu tun, auch wenn wir nicht die vollen 16 Bit zur Darstellung brauchen.

Der Wert 1024 entspricht dem binären Wert %0000 0100 0000 0000 bzw. dem hexadezimalen \$0400, das niederwertige Byte entspricht also dem Wert 0 und das höherwertige Byte dem Wert 4.

In der nachfolgenden Tabelle sind die 16 Bits von rechts beginnend von 0 bis 15 durchnummerniert dargestellt.

Die Bits 0 bis 7 bilden das niederwertige Byte und die Bits 8 bis 15 bilden das höherwertige Byte.

| Höherwertiges Byte | | | | | | | | Niederwertiges Byte | | | | | | | |
|--------------------|----|----|----|----|----|---|---|---------------------|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Die CPU liest die beiden Bytes von 16 Bit Zahlen in umgekehrter Reihenfolge aus dem Arbeitsspeicher, das heißt wir müssen zuerst das niederwertige Byte und dann das höherwertige Byte in den Speicher schreiben, damit die richtige Zahl eingelesen wird.

Das niederwertige Byte des Wertes 1024 entspricht, wie bereits vorhin erwähnt, dem Wert 0 und das höherwertige Byte entspricht dem Wert 4, d.h. wir müssen zuerst die 0 und darauffolgend die 4 im Speicher ablegen.

Da wir den Befehlscode 142 ja bereits in den Speicher geschrieben haben, müssen wir nur noch die beiden Teile der Speicheradresse 1024 in den Speicher schreiben.

In die Speicherstelle 5379 schreiben wir das niederwertige Byte, also den Wert 0.

POKE 5379,0

In die nächste Speicherstelle 5380 schreiben wir das höherwertige Byte, also den Wert 4.

POKE 5380,4

Nun fehlt uns nur noch der Befehl zur Beendigung unseres Programms, welcher, wie wir bereits wissen, den Befehlscode 96 hat.

Daher schreiben wir diesen Befehlscode noch in die Speicherstelle 5381:

POKE 5381,96

Unser kleines Maschinenprogramm steht nun startklar im Speicher.

Bevor wir es jedoch starten, gehen wir unser Programm nochmal Byte für Byte durch.

| Adresse | Inhalt | Beschreibung |
|---------|--------|---|
| 5376 | 162 | Befehlscode zum Laden des X Registers |
| 5377 | 4 | Gewünschter Inhalt des X Registers |
| 5378 | 142 | Befehlscode zum Schreiben des Inhalts des X Registers an eine bestimmte Speicheradresse |
| 5379 | 0 | Zuerst das niederwertige Byte der Speicheradresse 1024 ablegen |
| 5380 | 4 | Und dann das höherwertige Byte der Speicheradresse 1024 |
| 5381 | 96 | Befehlscode zum Beenden des Programms (Rückkehr zu BASIC) |

Nun wollen wir unser Programm starten. Löschen Sie dazu zunächst den Bildschirm durch drücken von SHIFT + CLR / HOME und bewegen den Cursor um eine oder mehrere Zeilen nach unten, sodass die linke, obere Ecke des Bildschirms frei wird.

Nach dem Start unseres Programms durch den Befehl SYS 5376 müsste in der linken, oberen Ecke des Bildschirms ein D zu sehen sein (siehe folgender Screenshot)



So programmiert man also in Maschinensprache, man schreibt Befehlscodes und Parameter in den Speicher. Anschließend wird das Maschinenprogramm durch den Befehl

SYS + Angabe der Startadresse des Maschinenprogramms

gestartet.

1.3.3 Erstellung von BASIC-Loadern

Bisher haben wir unsere Maschinenprogramme manuell durch POKE-Befehle im Direktmodus in den Speicher geschrieben. Wenn wir unsere Maschinenprogramme jedoch auf Diskette speichern wollen, um sie immer wieder laden und ausführen zu können, dann müssen wir einen anderen Weg beschreiten.

Dieser Weg führt über sogenannte BASIC-Loader. Das sind ganz normale BASIC-Programme, deren einzige Aufgabe es ist, ein Maschinenprogramm in den Speicher zu schreiben.

Dafür stehen uns zwei Möglichkeiten zur Verfügung und wir werden diese nachfolgend auch kennenlernen.

Die erste Möglichkeit haben wir vom Prinzip her bereits genutzt. Wir haben unsere Programme Byte für Byte durch POKE-Befehle in den Speicher geschrieben. Dies haben wir jedoch im Direktmodus durchgeführt und konnten unsere Programme daher nicht auf Diskette speichern.

Es geht also darum, die POKE-Befehle in ein BASIC-Programm zu verpacken, das man dann, so wie jedes andere Programm auch, auf Diskette speichern und jederzeit wieder laden bzw. ausführen kann.

Um den Sachverhalt zu veranschaulichen, möchte ich Ihnen zeigen, wie die BASIC-Loader für unsere bisherigen Maschinenprogramme aussehen würden.

Im ersten Programm haben wir das X Register mit dem Wert 200 geladen.

Der zugehörige BASIC-Loader würde so aussehen:

```
10 POKE 5376,162  
20 POKE 5377,200  
30 POKE 5378,96
```

Im zweiten Programm haben wir den Buchstaben D in die linke, obere Ecke des Bildschirms geschrieben.

Der zugehörige BASIC-Loader würde so aussehen:

```
10 POKE 5376,162  
20 POKE 5377,4  
30 POKE 5378,142  
40 POKE 5379,0  
50 POKE 5380,4
```

60 POKE 5381,96

Wenn man diese Programme mit RUN startet, wird das Maschinenprogramm Byte für Byte in den Speicher geschrieben. Das war bisher zwar auch der Fall, aber der Vorteil besteht eben darin, dass man das Programm auf Diskette speichern kann und nicht immer wieder neu eingeben muss.

Wenn der BASIC-Loader durchgelaufen ist, steht das Maschinenprogramm im Speicher und der BASIC-Loader selbst wird nicht mehr benötigt. Er kann daher mit NEW aus dem Speicher gelöscht werden. Sie haben solche BASIC-Programme sicher schon oft in abgedruckter Form in Zeitschriften wie der 64er gesehen.

Der Nachteil dieser Methode ist jedoch, dass der Schreibaufwand relativ hoch ist. Ein weiterer Nachteil besteht darin, dass es sehr aufwändig ist, die Startadresse des Programms zu ändern. Man müsste in jedem einzelnen POKE-Befehl die Speicheradresse entsprechend anpassen.

Daher bedient man sich im Regelfall einer anderen Möglichkeit, das Maschinenprogramm in den Speicher zu befördern.

Dazu legt man die Bytes des Maschinenprogramms in DATA-Zeilen ab und schreibt diese mittels eines einzelnen POKE-Befehls, welcher in einer FOR-Schleife ausgeführt wird, in den Speicher. Der Schreibaufwand wird dadurch deutlich geringer, weil der POKE-Befehl für jedes einzelne Byte entfällt.

Nachfolgend sehen Sie die beiden vorhin vorgestellten BASIC-Loader, wenn man sie entsprechend dieser Methode anpasst.

Hier der abgewandelte BASIC-Loader für das erste Programm:

```
10 FOR A=5376 TO 5378  
20 READ B  
30 POKE A,B  
40 NEXT A  
50 END  
60 DATA 162, 200,96
```

Und hier jener für das zweite Programm:

```
10 FOR A=5376 TO 5381  
20 READ B  
30 POKE A,B  
40 NEXT A  
50 END  
60 DATA 162, 4, 142, 0, 4, 96
```

Bei diesen einfachen Programmen fällt der verringerte Schreibaufwand noch nicht so auf, aber bei umfangreicheren Programmen wird der Effekt deutlicher. Stellen Sie sich vor, Sie haben ein Maschinenprogramm erstellt, welches aus 1000 Bytes besteht.

Dann wären das 1000 POKE-Befehle, die Sie eingeben müssten. Durch die Ablage der Bytes in den DATA-Zeilen erspart man sich die Eingabe dieser vielen POKE-Befehle.

Abgesehen davon, ist man bei dieser Methode flexibler in Bezug auf die Adresse, ab der das Maschinenprogramm in den Speicher geschrieben werden soll.

Der Startwert für die FOR-Schleife (hier 5376) entspricht der Startadresse des Programms im Speicher und der Endwert ergibt sich, indem man die **um 1 verminderte** Anzahl der Bytes zur Startadresse hinzuzählt. Dies ist dann die Adresse, an die das letzte Byte des Programms geschrieben wird.

In der DATA-Zeile haben wir hier 6 Bytes, das erste Byte (162) wird an die Adresse 5376 geschrieben und das letzte Byte (96) an die Adresse $5376 + 5$, also an die Adresse 5381.

Angenommen, Sie möchten, dass das Maschinenprogramm nicht mehr ab Adresse 5376, sondern ab Adresse 5500 in den Speicher geschrieben wird.

Dann bräuchten Sie nur den Kopf der FOR-Schleife folgendermaßen ändern:

FOR A=5500 TO 5505

Falls sich die Anzahl der Bytes in Ihrem Maschinenprogramm ändert, das Programm also kürzer oder länger wird, dann müssen Sie darauf achten, den Endwert in der FOR-Schleife entsprechend anzupassen.

Angenommen, das Maschinenprogramm wächst von 6 Bytes auf eine Länge von 53 Bytes. Dann entspricht das 53 Bytes in den DATA-Zeilen und der Endwert der FOR-Schleife müsste auf den Wert $5500 + 53 - 1$, also auf den Wert 5552 angepasst werden.

Der Kopf der FOR-Schleife würde dann also so aussehen:

FOR A=5500 TO 5552

Genau diese BASIC-Loader waren es, die mir damals immer ein Rätsel waren, weil ich die Bedeutung der vielen Zahlen nicht kannte.

Nun haben wir auch dieses Rätsel gelöst!

In den verbleibenden Beispiel-Programmen in diesem Kapitel werde ich jedoch der besseren Verständlichkeit wegen bei der vorherigen Methode bleiben, also die Bytes weiterhin mit separaten POKE-Befehlen in den Speicher schreiben. Die Programme sind relativ kurz und daher hält sich auch der Schreibaufwand in erträglichen Grenzen.

Das soll Sie jedoch nicht daran hindern, als Übung den entsprechenden BASIC-Loader zu schreiben :)

1.3.4 Laden des X Registers aus unterschiedlichen Quellen

Als nächstes möchte ich Ihnen eine sehr wichtige Gegebenheit in Bezug auf die Befehlscodes verdeutlichen, welche ich bisher verschwiegen habe, um am Anfang unnötige Verwirrung zu vermeiden.

Angenommen, wir wollen folgende zwei Aufgaben durchführen:

- Das X Register mit dem Wert 123 laden
- Das X Register mit dem Inhalt der Speicherstelle 1234 laden

Wie würde die Zahlenfolge für den ersten Befehl lauten? Richtig -> 162 123

Aber wie sieht es bei dem zweiten Befehl aus? Hier müssen wir die Speicheradresse 1234 wieder, wie vorhin, in das niederwertige und höherwertige Byte zerlegen.

1234 in hexadezimaler Form lautet \$04D2, das niederwertige Byte lautet \$D2 (dezimal 210) und das höherwertige Byte \$04 (dezimal 4).

Die Zahlenfolge für den zweiten Befehl lautet also 162 210 4 oder?

Sieht auf den ersten Blick korrekt aus, ist es aber nicht.

Denn die beiden Befehle tun zwar im Grunde dasselbe, nämlich das X Register mit einem Wert zu laden, aber die Quelle, aus der dieser Wert stammt, ist eine völlig andere.

Im ersten Fall ist direkt der Zahlenwert gemeint, der auf den Befehlscode folgt, nämlich der dezimale Wert 123.

Im zweiten Fall ist jedoch der Inhalt der Speicherstelle mit der Nummer 1234 gemeint. Hier kommt noch hinzu, dass die Adresse dieser Speicherstelle ja nicht nur aus einem Byte, sondern aus zwei Bytes besteht.

Im ersten Fall muss die CPU also das Byte, welches dem Befehlscode folgt, aus dem Arbeitsspeicher lesen und dieses in das X Register schreiben.

Im zweiten Fall muss die CPU nun zwei Bytes, welche dem Befehlscode folgen, aus dem Arbeitsspeicher lesen, diese beiden Bytes zu einer 16 Bit Speicheradresse verknüpfen (1234 in diesem Fall hier), dann den Inhalt an dieser Speicheradresse lesen und in das X Register schreiben.

Es passiert also eine ganze Menge mehr als beim ersten Befehl.

Der erste Befehl besteht also aus 2 Bytes (Befehlscode + Zahlenwert) und der zweite Befehl besteht aus 3 Bytes (Befehlscode + niederwertiges Byte der Adresse + höherwertiges Byte der Adresse)

Aber wie teilt man der CPU nun mit, dass man zwar das X Register mit einem Wert laden will, dieses mal aber nicht einen Zahlenwert als Parameter angibt, sondern eine Speicheradresse?

Die Lösung dieses Problems liegt in der Angabe eines anderen BefehlsCodes, welcher bewirkt, dass die CPU nicht nur ein Byte nach dem Befehlscode liest und dieses Byte in das X Register lädt, sondern zwei Bytes liest, diese beiden Bytes zu einer Speicheradresse zusammensetzt, den Inhalt an dieser Speicheradresse ausliest und in das X Register lädt.

In diesem Fall ist es der Befehlscode 174 anstelle von 162.

Trifft die CPU also auf den Befehlscode 174, lädt sie zwar das X Register mit einem Wert, aber sie tut etwas völlig anderes, um an diesen Wert zu kommen als beim Befehlscode 162.

Wenn wir nun beim zweiten Befehl den Befehlscode 162 durch den Code 174 ersetzen, dann läuft unser Programm richtig, weil die CPU den Befehl nun genau so interpretiert, wie wir ihn gemeint haben.

Würden wir es beim Befehlscode 162 belassen, dann arbeitet unser Programm falsch, weil die CPU den Wert 210 in das X Register laden würde und den Wert 4 bereits als den nächsten Befehlscode interpretiert!

Solche unterschiedlichen Befehlscodes gibt es bei vielen Maschinenbefehlen, um mit den verschiedenen Bedeutungen der Parameter umgehen zu können.

Sogar beim soeben verwendeten Befehl zum Laden des X Registers gibt es über die Befehlscodes 162 und 174 hinaus noch einige weitere.

Einen davon möchte ich Ihnen noch vorstellen, da er sehr gut zum zweiten Befehl passt, der den Inhalt der Speicherstelle 1234 in das X Register geladen hat.

Wie bereits erwähnt, ist 1234 ein 16 Bit Wert, benötigt also zwei Bytes im Speicher. Dies gilt jedoch nicht für die Speicheradressen zwischen 0 und 255, denn hier reicht ein einziges Byte für die Darstellung aus.

Will man nun den Inhalt einer Speicherstelle auslesen, deren Adresse zwischen 0 und 255 liegt, dann würde doch ein Byte für die Adressangabe ausreichen, weil das höherwertige Byte in diesen Fällen ja immer 0 enthält.

Am Beispiel der Adresse 200 würde die Zahlenfolge unter Verwendung der 16 Bit Adresse 174 200 0 lauten.

Das funktioniert natürlich wunderbar, allerdings verschenkt man hier durch die überflüssige 0 ein Byte an Speicherplatz.

Da wäre es doch toll, wenn es einen Befehlscode gäbe, der ebenso wie der Befehlscode 162 nur ein Byte nach dem Befehlscode liest, dieses aber nicht als Zahl interpretiert, welche in das X Register geladen werden soll, sondern als 8 Bit Speicheradresse zwischen 0 und 255 und deren Inhalt in das X Register lädt.

Und natürlich gibt es diesen Befehlscode, er lautet 166 und verkürzt die obige Zahlenfolge auf 166 200.

Das Programm wird dadurch nicht nur kürzer, sondern auch schneller, weil die CPU nicht mehr zwei Werte (200 und 0), sondern nur mehr einen Wert (200) aus dem Arbeitsspeicher lesen muss.

Die Einsparung mag vielleicht bei diesem einen Befehl noch nicht so sehr ins Gewicht fallen, aber sobald Befehle wiederholt innerhalb von Schleifen ausgeführt werden, kann die Einsparung je nach Befehl durchaus deutliche Auswirkungen auf die Geschwindigkeit haben.

Sie sehen also, dass man auf der Ebene der Maschinensprache die Kontrolle über jedes einzelne Byte hat und auf diese Weise Speicherverbrauch und die Ausführungsgeschwindigkeit optimieren kann.

1.3.5 Programm 3: Einfache Rechenoperationen (Version 1)

Als nächstes möchte ich Ihnen zeigen, wie man ein paar einfache Rechenoperationen in Maschinensprache umsetzt.

Was soll das Programm machen?

- Den Akkumulator mit dem Wert 100 laden
- Zum Inhalt des Akkumulators den Wert 75 addieren
- Den Inhalt des X Registers mit dem Wert 52 laden
- Den Inhalt des X Registers um 1 erhöhen
- Den Inhalt des Y Registers mit dem Wert 37 laden
- Den Inhalt des Y Registers um 1 vermindern
- Programm beenden

Für jeden dieser Schritte gibt es eigene Maschinenbefehle.

Der Befehlscode zum Laden des Akkumulators mit einem Zahlencode lautet 169, d.h. die Zahlenfolge für den ersten Befehl lautet 169 100.

Der Befehlscode um einen Wert zum Inhalt des Akkumulators zu addieren lautet 105, d.h. die Zahlenfolge für den zweiten Befehl lautet 105 75.

Der Befehlscode, welcher das X Register mit einem Wert lädt, lautet wie bereits bekannt 162, d.h. die Zahlenfolge für den dritten Befehl lautet 162 52.

Der Befehlscode, welcher das X Register um 1 erhöht, lautet 232, d.h. die Zahlenfolge für den vierten Befehl besteht nur aus 232, da dieser Befehl keine Parameter benötigt.

Der Befehlscode, welcher das Y Register mit einem Wert lädt, lautet 160, d.h. die Zahlenfolge für den fünften Befehl lautet 160 37.

Der Befehlscode, welcher das Y Register um eins vermindert, lautet 136, d.h. die Zahlenfolge für den sechsten Befehl besteht nur aus 136, da dieser Befehl keine Parameter benötigt.

Der Befehlscode, welcher das Programm beendet und zu BASIC zurückkehrt, lautet wie bereits bekannt 96, d.h. die Zahlenfolge für den letzten Befehl besteht nur aus 96, da auch dieser Befehl keine Parameter benötigt.

Zusammengefasst lautet die Zahlenfolge für unser Maschinenprogramm also:

169, 100, 105, 75, 162, 52, 232, 160, 37, 136, 96

Schreiben wir also wieder mit POKE-Anweisungen diese Zahlenfolge in den Speicher. Da wir dieses mal etwas mehr Zeilen haben, werden wir die POKE-Anweisungen wie vorhin in ein eigenes BASIC-Programm verpacken, das wir auf Diskette speichern und wieder laden können.

Ich habe nämlich im Anschluß noch eine kleine Herausforderung auf Lager :)

```
10 POKE 5376,169
20 POKE 5377,100
30 POKE 5378,105
40 POKE 5379,75
50 POKE 5380,162
60 POKE 5381,52
70 POKE 5382,232
80 POKE 5383,160
90 POKE 5384,37
100 POKE 5385,136
110 POKE 5386,96
```

Speichern Sie dieses Programm mit SAVE „CALC1“,8 auf Diskette ab.

Starten Sie das Programm mit RUN, um das Maschinenprogramm in den Speicher zu schreiben und starten Sie selbiges mit SYS 5376. Das Maschinenprogramm läuft durch und der C64 meldet sich wieder mit READY.

Wenn alles korrekt abgelaufen ist, müsste der Akkumulator nun den Wert 175, das X Register den Wert 53 und das Y Register den Wert 36 beinhalten.

Ob dies der Fall ist, können wir über die bereits genannten Speicherstellen 780 (für den Akkumulator), 781 (für das X Register) und 782 (für das Y Register) herausfinden:

Und wie auf dem folgenden Screenshot zu sehen ist, hat alles wunderbar funktioniert, denn alle drei Register enthalten den korrekten Wert.

```
78 POKE 5382,232
80 POKE 5383,160
90 POKE 5384,37
100 POKE 5385,136
110 POKE 5386,96
READY.
RUN
READY.
SYS 5376
READY.
PRINT PEEK (780)
175
READY.
PRINT PEEK (781)
53
READY.
PRINT PEEK (782)
36
READY.
```

1.3.6 Programm 4: Einfache Rechenoperationen (Version 2)

Doch nun zu der kleinen Herausforderung, welche ich vorhin angesprochen habe.

Nehmen wir mal an, wir könnten die Inhalte dieser drei Register nicht über die Speicherstellen 780, 781 und 782 abfragen.

Von BASIC aus haben wir ansonsten keinen direkten Zugriff auf die CPU-Register.

Um dennoch an die aktuellen Werte zu kommen, können wir unser Maschinenprogramm so verändern, dass es die Inhalte des Akkumulators, des X Registers und des Y Registers irgendwo im Arbeitsspeicher ablegt, denn diesen können wir von BASIC aus erreichen und Inhalte von Speicherstellen auslesen.

Anbieten würden sich die Speicherstellen gleich hinter dem Ende des Programms, also in die drei Bytes, welche auf den Befehlscode 96 folgen.

Mit dem Befehl PEEK können wir, nachdem das Programm durchgelaufen ist, den Inhalt dieser drei Speicherstellen dann direkt auslesen.

Das alles klingt komplizierter als es ist, es stellt jedoch eine sehr gute Übung dar und vertieft das Verständnis in Bezug auf die Maschinensprache.

Keine Sorge, dies wird unser letztes Programm in Maschinensprache sein, denn nun sind Sie fit für die Assemblersprache. Also, einmal noch durchhalten und die qualvolle Programmierung in Maschinensprache hat endlich ein Ende.

Also, ran ans Werk!

Was ist zu tun? Wir müssen den Inhalt des Akkumulators, des X Registers und des Y Registers in drei Speicherstellen schreiben, welche im Speicher direkt an unser Programm anschließen, also direkt auf den Befehlscode 96 folgen.

Welche Speicherstellen das sind, müssen wir noch ermitteln.

Um den Wert des Akkumulators in eine Speicherstelle zu schreiben, brauchen wir den Maschinenbefehl mit dem Befehlscode 141. Darauf wird im Speicher dann die aus zwei Bytes bestehende Speicheradresse folgen, an die der Inhalt des Akkumulators geschrieben wird.

Unser Programm wächst also um 3 Bytes.

Der nächste Wert, den wir in den Speicher schreiben, ist der Befehlscode, um den Inhalt des X Registers in eine Speicherstelle zu schreiben. Dieser lautet 142 und auf diesen wird im Speicher dann die aus zwei Bytes bestehende Speicheradresse folgen, an die der Inhalt des X Registers geschrieben wird.

Unser Programm wächst wiederum um 3 Bytes.

Weiter geht's mit dem Befehlscode, um den Inhalt des Y Registers in eine Speicherstelle zu schreiben. Er lautet 140 und auf diesen wird im Speicher dann die aus zwei Bytes bestehende Speicheradresse folgen, an die der Inhalt des Y Registers geschrieben wird.

Unser Programm wächst erneut um 3 Bytes, d.h. wir benötigen im Speicher 9 zusätzliche Bytes um die soeben beschriebenen Befehle im Speicher unterzubringen.

Am Ende fehlt dann noch ein Byte für den Befehlscode 96 zur Beendigung unseres Programms.

Insgesamt benötigen wir im Speicher also 9 zusätzliche Bytes für unser Programm.

Nun wollen wir all diese Befehle in unser Programm einfügen.

Die Rechenoperationen enden an der Speicherstelle 5385 mit dem Befehl zum Vermindern des Y Registers. Darauf folgt direkt an der Speicherstelle 5386 der Befehlscode 96 zum Beenden des Programms (siehe Zeilen 100 und 110 im obigen BASIC-Programm)

Unsere neuen Befehle fügen wir ab der Speicherstelle 5386 ein.

Dazu fügen wir ab Zeile 110 zehn neue Zeilen mit POKE-Befehlen ein.

In Zeile 110, 140 und 170 werden die vorhin genannten Befehlscodes in den Speicher geschrieben.

Die anderen POKE-Befehle können wir erst vervollständigen, wenn wir wissen, welche Speicheradressen wir dort eintragen müssen.

```

110 POKE 5386,141
120 POKE 5387,
130 POKE 5388,
140 POKE 5389,142
150 POKE 5390,
160 POKE 5391,
170 POKE 5392,140
180 POKE 5393,
190 POKE 5394,
200 POKE 5395,96

```

Durch das Einfügen der neuen Befehle und deren Parametern hat sich das Ende unseres Programms an die Speicherstelle 5395 verschoben.

Dadurch wissen wir nun auch, an welche Speicherstellen wir die drei Werte schreiben müssen, nämlich an die drei Speicherstellen ab Adresse 5396.

Der Inhalt des Akkumulators kommt an die Speicherstelle 5396, der Inhalt des X Registers an die Speicherstelle 5397 und der Inhalt des Y Registers an die Speicherstelle 5398.

Um diese 16 Bit Adressen in unser Maschinenprogramm eintragen zu können, brauchen wir jeweils die niederwertigen Bytes und die höherwertigen Bytes.

Diese Werte ermitteln wir am leichtesten, indem wir die hexadezimale Form der Adressen ermitteln.

| Adresse | Hexadezimal | Niederwertiges Byte (hexadezimal) | Höherwertiges Byte (hexadezimal) | Niederwertiges Byte (dezimal) | Höherwertiges Byte (dezimal) |
|---------|-------------|-----------------------------------|----------------------------------|-------------------------------|------------------------------|
| 5396 | \$1514 | \$14 | \$15 | 20 | 21 |
| 5397 | \$1515 | \$15 | \$15 | 21 | 21 |
| 5398 | \$1516 | \$16 | \$15 | 22 | 21 |

Nun können wir in unserem BASIC-Loader die niederwertigen und höherwertigen Bytes der Adressen eintragen (zur besseren Veranschaulichung habe ich diese Werte grün markiert)

```

110 POKE 5386,141
120 POKE 5387,20
130 POKE 5388,21
140 POKE 5389,142
150 POKE 5390,21
160 POKE 5391,21
170 POKE 5392,140
180 POKE 5393,22
190 POKE 5394,21

```

Speichern Sie nun dieses geänderte BASIC-Programm mit SAVE „CALC2“,8

Hier noch einmal zum Verständnis eine Beschreibung der Bytes, die wir neu eingefügt haben, sowie am Ende gelb hinterlegt die drei Bytes hinter dem Ende des Programms, welche nach Beendigung des Programms die kopierten Registerwerte beinhalten.

| Adresse | Inhalt | Beschreibung |
|---------|--------|--|
| 5386 | 141 | Befehlscode zum Speichern des Inhalts des Akkumulators in einer Speicherstelle |
| 5387 | 20 | Niederwertiges Byte der Adresse 5396 |
| 5388 | 21 | Höherwertiges Byte der Adresse 5396 |
| 5389 | 142 | Befehlscode zum Speichern des Inhalts des X Registers in einer Speicherstelle |
| 5390 | 21 | Niederwertiges Byte der Adresse 5397 |
| 5391 | 21 | Höherwertiges Byte der Adresse 5397 |
| 5392 | 140 | Befehlscode zum Speichern des Inhalts des Y Registers in einer Speicherstelle |
| 5393 | 22 | Niederwertiges Byte der Adresse 5398 |
| 5394 | 21 | Höherwertiges Byte der Adresse 5398 |
| 5395 | 96 | Befehlscode zum Beenden des Programms |
| 5396 | 175 | Wert aus Akkumulator |
| 5397 | 53 | Wert aus X Register |
| 5398 | 36 | Wert aus Y Register |

Starten Sie nun das Programm mit RUN, damit unser Maschinenprogramm in den Speicher übertragen wird und starten dieses mit SYS 5376.

Sobald die Meldung READY wieder erscheint, geben Sie folgende Befehle ein:

```
PRINT PEEK (5396)
PRINT PEEK (5397)
PRINT PEEK (5398)
```

Wenn Sie dieselben Ergebnisse erhalten, welche auf dem folgenden Screenshot zu sehen sind, dann haben Sie alles richtig gemacht!

The screenshot shows a terminal window on a Commodore 64. The screen displays assembly language code being executed and memory dump results. The code includes POKE commands to memory locations 5391, 5392, 5393, 5394, and 5395, followed by a series of READY. messages and the RUN command. Subsequent lines show the results of PRINT PEEK commands for addresses 5396, 5397, and 5398, returning values 175, 53, and 36 respectively. The background of the terminal window is light blue.

```
160 POKE 5391,21
170 POKE 5392,140
180 POKE 5393,22
190 POKE 5394,21
200 POKE 5395,96
READY.
RUN
READY.
SYS 5376
READY.
PRINT PEEK (5396)
175
READY.
PRINT PEEK (5397)
53
READY.
PRINT PEEK (5398)
36
READY.
```

OK, soweit so gut. So programmiert man also in Maschinensprache! Zugegeben, sehr mühsam sich diese Befehlscodes zusammenzusuchen und gemeinsam mit den Parametern Byte für Byte in den Speicher zu schreiben. Doch die Rettung ist nah und lautet Assembler :)

2 Programmieren in Assembler

2.1 Warum Assembler?

In diesem Kapitel will ich Ihnen die Unterschiede zwischen der Maschinensprache und der Assemblersprache aufzeigen. Wir werden sehen, worin der Fortschritt gegenüber der Maschinensprache besteht und namentlich schon mal ein paar Assemblerbefehle kennenlernen.

Konkret an die Programmierung geht's dann jedoch erst im nächsten Kapitel, denn in diesem Kapitel will ich Sie zunächst einmal schrittweise von der Maschinensprache zur Assemblersprache überleiten, damit Sie ein Gefühl dafür bekommen, wie so ein Assemblerprogramm überhaupt aussieht.

Eines kann ich Ihnen jedoch schon mal versichern:

Nachdem ich Sie nun durch die harte Schule der Programmierung in Maschinensprache gejagt habe, wird Ihnen die Programmierung in der Assemblersprache wie ein Wellness-Urlaub erscheinen.

Die kleinen Programme, die wir bisher erstellt haben, waren reine Maschinenprogramme, d.h. eine Folge von Zahlen, welche die CPU eine nach der anderen verarbeitet.

Alle Programme, egal in welcher Programmiersprache sie geschrieben sind, müssen von einem zusätzlichen Programm in diese Maschinensprache übersetzt werden, da die CPU nur diese Folge von Zahlen "versteht".

Mit Befehlen wie PRINT oder GOTO kann sie nichts anfangen, ebenso wenig mit Texten oder Variablennamen. Alles muss letztendlich in eine Folge von Zahlen übersetzt werden, welche die CPU dann verarbeiten kann.

Bei den sogenannten höheren Programmiersprachen wie BASIC, C, PASCAL und wie sie alle heißen, übernimmt diese Übersetzung ein Interpreter oder Compiler.

Im Falle der Assemblersprache heißt dieses Übersetzungsprogramm verwirrenderweise ebenfalls Assembler, sodass man in Gesprächen oft zusätzlich erwähnen muss, ob nun die Sprache oder das Übersetzungsprogramm gemeint ist.

Es gibt viele solcher Assembler-Übersetzungsprogramme, die sich in ihrem Funktionsumfang und der Bedienung teilweise sehr unterscheiden. In diesem Buch verwenden wir beispielsweise die Assembler-Übersetzungsprogramme SMON und später dann Turbo Macro Pro.

Wir werden in diesem Kapitel auch selbst mal in die Rolle des Übersetzers schlüpfen und ein Assembler-Programm per Hand Schritt für Schritt in Maschinencode übersetzen, den die CPU dann ausführen kann.

Was sind nun die Unterschiede zwischen Maschinensprache und Assembler?

Maschinensprache ist, wie bereits erwähnt, nur eine Folge von Zahlen, welche Maschinenbefehle mit ihren Parametern enthält. Diese Folge von Zahlen haben wir durch POKE-Befehle direkt in den Speicher geschrieben und das Maschinenprogramm dann mit dem Befehl SYS gestartet.

Die CPU hat mit dieser Folge von Zahlen keine Verständnisprobleme mehr, aber wir Menschen tun uns sehr schwer damit, weil wir ja nicht in Zahlen denken, sondern eher in Worten.

Außerdem ist diese Form der Programmierung selbst bei kleinen Programmen mühsam, zeitaufwendig und vor allem fehleranfällig.

Größere Programme sind auf diese Art und Weise nicht umsetzbar, von nachträglichen Änderungen bzw. Korrekturen ganz zu schweigen.

Und da sind wir schon beim ersten Unterschied zur Maschinensprache. In Assembler werden die Maschinenbefehle nicht mehr durch Befehlscodes in Form von Zahlen angegeben, sondern durch einfache Befehle in Textform (sogenannte Mnemonics).

Aus diesen Befehlen kann man verhältnismäßig leicht (und mit der Zeit bzw. zunehmender Übung auf den ersten Blick) ableiten, was der Befehl tut.

Bevor wir weitermachen, fassen wir doch mal zusammen, welche Aktionen wir bis jetzt in Maschinensprache durchgeführt haben und sehen uns an, wie man diese Maschinenbefehle in der Assembler-Sprache formuliert.

Wir haben gelernt, wie man

1. den Akkumulator, das X Register und das Y Register mit einem Wert lädt
2. den Inhalt des Akkumulators, des X Registers und des Y Registers an eine 8bit Speicheradresse schreibt
3. den Inhalt des Akkumulators, des X Registers und des Y Registers an eine 16bit Speicheradresse schreibt
4. den Inhalt des X Registers und des Y Registers um 1 erhöht bzw. um 1 vermindert
5. einen Wert zum Inhalt des Akkumulators addiert
6. ein Maschinenprogramm beendet und zu BASIC zurückkehrt

Wir haben echte Maschinenprogramme geschrieben, in dem wir mit POKE-Befehlen die BefehlsCodes und deren Parameter direkt in den Speicher geschrieben haben.

Doch wie bereits vorhin erwähnt, ist diese Art der Programmierung nicht praktikabel und daher kommen wir nun zur Programmierung in Assembler.

Der Hauptvorteil der Assemblersprache ist der, dass man sich nicht mehr mit BefehlsCodes rumschlagen muss. Denn wenn man in Maschinensprache programmiert, muss man sich für jeden einzelnen Befehl den jeweils passenden Befehlscode aus einer Liste heraussuchen, bevor man ihn in den Speicher schreiben kann.

Und wie wir bereits gelernt haben, hängt der Befehlscode auch von den verwendeten Parametern ab. Um das X Register direkt mit einem Zahlenwert zu laden, braucht man einen anderen Befehlscode, als wenn man es mit dem Inhalt einer Speicheradresse laden will. Und auch hier

macht es einen Unterschied in Bezug auf den Befehlscode, ob es sich um eine 8bit- oder eine 16bit-Adresse handelt.

Um in Assembler das X Register zu laden, verwendet man den Befehl LDX und dahinter gibt man den Parameter an. Das kann ein fixer Zahlenwert, eine 8bit- oder eine 16bit Speicheradresse sein. Unabhängig davon wird jedoch immer das Befehlswort LDX verwendet.

Doch wie weiß das Assembler-Übersetzungsprogramm nun, in welchen Befehlscode es den jeweiligen Befehl übersetzen soll?

Die Antwort:

Anhand der Kennzeichnung des Parameters. Meint man einen fixen Zahlenwert, dann schreibt man das Zeichen # davor. Ist hingegen eine Speicheradresse gemeint (egal ob es sich um eine 8bit- oder eine 16bit-Adresse) handelt, dann lässt man dieses Zeichen einfach weg.

Hier drei Beispiele:

| Befehl | Bedeutung |
|----------|---|
| LDX #200 | Lädt das X Register mit dem Wert 200 |
| LDX 100 | Lädt das X Register mit dem Inhalt der 8bit Speicheradresse 100 |
| LDX 1024 | Lädt das X Register mit dem Inhalt der 16bit Speicheradresse 1024 |

LDX #200

Das Übersetzungsprogramm erkennt anhand des Zeichens #, dass hier ein fixer Zahlenwert gemeint ist, der in das X Register geladen werden soll.

Der generierte Maschinencode lautet also: 162, 200

Wie kommt das Übersetzungsprogramm hier auf den Befehlscode 162? Indem es in einer Tabelle nachsieht, welchen Befehlscode es erzeugen muss, wenn man einen fixen Zahlenwert in das X Register laden will.

In dieser Tabelle ist für jede Verwendungsart eines Befehls der entsprechende Befehlscode vermerkt.

LDX 100

Das Übersetzungsprogramm erkennt, dass es sich bei dem Parameter 100 um eine Speicheradresse handelt, weil das Zeichen # nicht vorhanden ist. Außerdem wird erkannt, dass es sich um eine 8bit Speicheradresse handelt, da die Adresse im Bereich von 0 bis 255 liegt.

Der generierte Maschinencode lautet also: 166, 100

Wie kommt das Übersetzungsprogramm hier auf den Befehlscode 166? Indem es in der Tabelle nachsieht, welchen Befehlscode es erzeugen muss, wenn man das X Register mit dem Inhalt einer 8bit Adresse laden will.

LDX 1024

Das Übersetzungsprogramm erkennt, dass es sich bei dem Parameter 1024 um eine Speicheradresse handelt, weil das Zeichen # nicht vorhanden ist. Außerdem wird erkannt, dass es sich um eine 16bit Speicheradresse handelt, da die Adresse größer als 255 ist.

Der generierte Maschinencode lautet also: 174, 0, 4

Wie kommt das Übersetzungsprogramm hier auf den Befehlscode 174? Indem es in der Tabelle nachsieht, welchen Befehlscode es erzeugen muss, wenn man das X Register mit dem Inhalt einer 16bit Adresse laden will.

Und genau diese umständliche Suche entfällt, wenn man in Assembler programmiert.

Das Übersetzungsprogramm erkennt den Befehl LDX und sucht sich anhand der Art des Parameters den richtigen Befehlscode aus der Tabelle.

Es nimmt uns auch bei der Angabe von Speicheradressen die Arbeit ab. Beim dritten Befehl beispielsweise wird die Adresse 1024 in das niederwertige und höherwertige Byte zerlegt und diese in der richtigen Reihenfolge (zuerst das niederwertige, dann das höherwertige Byte) in den Speicher geschrieben.

In Maschinensprache mussten wir selbst dafür sorgen, dass die Bytes in der richtigen Reihenfolge im Speicher stehen.

Um anstatt des X Registers den Akkumulator zu laden, verwendet man statt dem Befehlswort LDX einfach das Befehlswort LDA und im Falle des Y Registers das Befehlswort LDY.

Umgelegt auf den Akkumulator und das Y Register würden die obigen drei Befehle also folgendermaßen lauten:

Akkumulator

LDA #200
LDA 100
LDA 1024

Y Register

LDY #200
LDY 100
LDY 1024

Auch bei den Befehlen LDA und LDY verhält es sich gleich. Bei unterschiedlichen Parametern (fixer Zahlenwert oder Speicheradresse) wird jeweils ein anderer Befehlscode erzeugt.

Bevor wir unsere Reise fortsetzen, muss ich Ihnen noch eine wichtige Information mit auf den Weg geben.

Der C64 „versteht“ nur eine einzige Maschinensprache, nämlich jene der CPU MOS 6510.

Im Gegensatz dazu gibt es jedoch mehrere Übersetzungsprogramme (ebenfalls Assembler genannt), die unseren Assemblercode in die Maschinensprache des C64 übersetzen.

Wir werden zunächst das Programm SMON verwenden und wechseln dann auf das Programm Turbo Macro Pro, da dieses komfortabler zu bedienen und für umfangreichere Assembler-Programme besser geeignet ist.

Wichtig für Sie als angehender Assembler-Programmierer ist, dass der SMON Zahlenwerte anders interpretiert als beispielsweise der Turbo Macro Pro.

SMON interpretiert alle Zahlenwerte als hexadezimale Zahlen. Der Turbo Macro Pro hingegen interpretiert Zahlenwerte nur dann als hexadezimale Zahlen, wenn man ihnen das Zeichen \$ voranstellt. Das gilt ebenso für Speicheradressen.

Um dies zu veranschaulichen und die unterschiedliche Schreibweise darzustellen, möchte ich die drei Befehle von vorhin wieder heranziehen.

| Befehl | SMON | Turbo Macro Pro |
|---|--------------------------|--------------------------|
| Lade das X Register mit dem fixen Zahlenwert 200 | LDX #\$C8 oder LDX #C8 | LDX #\$C8 oder LDX 200 |
| Lade das X Register mit dem Inhalt der 8bit Speicheradresse 100 | LDX \$64 oder LDX 64 | LDX \$64 oder LDX 100 |
| Lade das X Register mit dem Inhalt der 16bit Speicheradresse 1024 | LDX \$0400 oder LDX 0400 | LDX \$0400 oder LDX 1024 |

Wie hier in der Tabelle zu sehen ist, kann man bei der Arbeit mit SMON das Zeichen \$ weglassen, da sowieso alle Werte als hexadezimale Werte interpretiert werden. Bei der Arbeit mit dem Turbo Macro Pro ist die Angabe des Zeichens \$ jedoch erforderlich, da dieser auch die Angabe von dezimalen Werten ermöglicht.

Was haben wir noch gelernt? Richtig, das Schreiben der Register-Inhalte an eine 8bit- oder 16bit Adresse.

Um den Inhalt des Akkumulators an eine Speicheradresse zu schreiben, verwenden wir das Befehlswort STA. Die entsprechenden Befehlsworte für das X Register und das Y Register lauten, wie nicht anders zu erwarten, STX und STY.

Auch hier hängt der generierte Maschinencode wieder davon ab, ob wir den Registerinhalt an eine 8bit- oder 16bit Speicheradresse schreiben wollen.

Mit folgendem Befehl schreiben wir den Inhalt des Akkumulators an die 8bit-Speicheradresse 100.

STA \$64 (sowohl für SMON als auch für Turbo Macro Pro)
STA 100 (nur Turbo Macro Pro)

Angewandt auf das X Register und das Y Register lauten diese Befehlen

STX \$64 bzw. STY \$64 (sowohl für SMON als auch für Turbo Macro Pro)
STX 100 bzw. STY 100 (nur Turbo Macro Pro)

Wenn wir den Registerinhalt an eine 16bit Speicheradresse schreiben wollen, dann ändert sich an der grundsätzlichen Schreibweise des Befehls nichts.

Mit folgendem Befehl schreiben wir den Inhalt des Akkumulators an die 16bit-Speicheradresse 1024.

STA \$0400 (sowohl für SMON als auch für Turbo Macro Pro)
STA 1024 (nur Turbo Macro Pro)

Angewandt auf das X Register und das Y Register lauten diese Befehle

STX \$0400 bzw. STY \$0400 (sowohl für SMON als auch für Turbo Macro Pro)
STX 1024 bzw. STY 1024 (nur Turbo Macro Pro)

Darüberhinaus haben wir noch gelernt, wie man die Inhalte des X Registers und des Y Registers um 1 erhöhen bzw. um 1 vermindern kann.

Die entsprechenden Befehle für das X Register lauten

INX (erhöht den Inhalt des X Registers um 1)
DEX (vermindert den Inhalt des X Registers um 1)

Für das Y Register gibt es für diese Zwecke folgende Befehle:

INY (erhöht den Inhalt des Y Registers um 1)
DEY (vermindert den Inhalt des Y Registers um 1)

Dann haben wir noch gelernt, wie man zum Inhalt des Akkumulators einen Wert addieren kann.

Das entsprechende Befehlswort lautet ADC.

Um also beispielsweise den Wert 56 zum Inhalt des Akkumulators zu addieren, wendet man folgende Befehle an:

ADC #\$38 (sowohl für SMON als auch für Turbo Macro Pro)
ADC 56 (nur Turbo Macro Pro)

Und zuguterletzt haben wir gelernt, wie man ein Maschinenprogramm beendet und zu BASIC zurückkehrt.

In Assembler gibt es dazu das Befehlswort RTS.

Nun möchte ich Ihnen anhand eines kleinen Assembler-Programms zeigen, wie dieses von einem Übersetzungsprogramm (z.B. SMON oder Turbo Macro Pro) in Maschinensprache übersetzt wird.

Da wir aktuell noch keinen SMON oder Turbo Macro Pro zur Verfügung haben, schlüpfen wir kurzerhand selbst in die Rolle des Übersetzers. Auf diese Art und Weise bekommen Sie ein gutes Gefühl dafür, wie der Weg vom Assembler-Programm zum ausführbaren Maschinenprogramm aussieht.

Hinweis:

Ich werde ab jetzt in den Assembler-Befehlen Zahlenwerte in hexadezimaler Schreibweise mit dem vorangestellten \$ - Zeichen angeben, denn auf diese Art und Weise interpretiert sowohl SMON als auch Turbo Macro Pro unser Programm richtig.

Nachfolgend sehen Sie das kleine Assembler-Programm, welches wir als „Eingabe“ für die Übersetzung entgegen nehmen.

```
LDA #$00  
STA $FB
```

```
LDX $FB  
INX  
STX $0400
```

```
LDY $FB  
INY  
INY  
STY $0401
```

```
RTS
```

Vorhin habe ich erwähnt, dass das Übersetzungsprogramm eine Tabelle mit sich führt, in der es nachsieht, welchen Befehlscode es für die Verwendungsart des jeweiligen Befehls erzeugen muss.

In dem Assembler-Programm verwenden wir die Befehle LDA, LDX, LDY, STA, STX, STY, INX, INY und RTS.

Damit wir dieses Assembler-Programm in Maschinensprache übersetzen können, bräuchten wir beispielsweise folgende Tabelleneinträge, damit wir die passenden Befehlscodes ermitteln können.

| Befehl | Verwendungsart | Zu erzeugender Befehlscode |
|--------|----------------------------------|----------------------------|
| LDA | Fixen Zahlenwert laden | 169 |
| LDX | Inhalt von 8bit Adresse laden | 166 |
| LDY | Inhalt von 8bit Adresse laden | 164 |
| STA | Inhalt an 8bit Adresse schreiben | 133 |

| Befehl | Verwendungsart | Zu erzeugender Befehlscode |
|--------|-----------------------------------|----------------------------|
| STX | Inhalt an 16bit Adresse schreiben | 142 |
| STY | Inhalt an 16bit Adresse schreiben | 140 |
| INX | | 232 |
| INY | | 200 |
| RTS | | 96 |

Beginnen wir also nun damit, dieses Assembler-Programm in Maschinensprache zu übersetzen.

LDA #\$00

Wir erkennen einen Text mit dem Inhalt LDA. Nun müssen wir zuerst einmal prüfen, ob dies überhaupt ein gültiger Assembler-Befehl ist. Wir sehen also in der ersten Spalte unserer Tabelle nach, ob wir den Befehl LDA dort finden und finden ihn auch gleich an erster Stelle ganz oben.

Nun stellt sich die Frage, welchen Befehlscode wir in diesem Fall erzeugen müssen. Sehen wir uns daher den Parameter an. Vor dem Zahlenwert steht das Zeichen #, d.h. die Person, welche das Assemblerprogramm geschrieben hat, will an dieser Stelle einen direkt angegebenen Zahlenwert in den Akkumulator laden.

Nun gehen wir alle Zeilen für den Befehl LDA durch und sehen in der zweiten Spalte nach, ob wir für die Verwendungsart „Fixen Zahlenwert laden“ einen Eintrag finden. In der ersten Zeile werden wir fündig und aus der dritten Spalte können wir nun den passenden Befehlscode auslesen.

In diesem Fall lautet er 169.

Der hinter dem Befehl LDA angegebene Zahlenwert lautet 0, d.h. der Befehl LDA #\$00 wird in die Zahlenfolge 169, 0 übersetzt. Diese Zahlenfolge notieren wir uns auf einem Zettel. Dieser enthält am Ende des Übersetzungsvorgangs die Zahlenfolge für das komplette Maschinenprogramm.

Nach der Übersetzung des ersten Befehls LDA #\$00 steht auf dem Zettel die Zahlenfolge 169, 0

Dieses Schema wenden wir nun auf alle restlichen Befehle an.

STA \$FB

Der Befehl STA ist gültig, da er in der ersten Spalte der Tabelle zu finden ist. Das Zeichen # ist nicht vorhanden, d.h. es folgt eine Speicheradresse als Parameter. Der hexadezimale Wert \$FB entspricht dem dezimalen Wert 251, d.h. die Speicheradresse ist eine 8bit-Adresse.

Nun gehen wir alle Zeilen für den Befehl STA durch und sehen in der zweiten Spalte nach, ob wir für die Verwendungsart „Inhalt an 8bit Adresse schreiben“ einen Eintrag finden. In der vierten Zeile werden wir fündig und aus der dritten Spalte können wir nun den passenden Befehlscode auslesen.

Der gefundene Befehlscode lautet 133 und die Speicheradresse lautet 251, d.h. der Befehl STA \$FB wird in die Zahlenfolge 133, 251 übersetzt. Diese beiden Zahlen notieren wir uns wieder auf dem oben erwähnten Zettel.

Dieser enthält zum aktuellen Zeitpunkt die Zahlenfolge 169, 0, 133, 251

LDX \$FB

Der Befehl LDX ist gültig, da er in der ersten Spalte der Tabelle zu finden ist. Wieder ist das Zeichen # nicht vorhanden, d.h. es folgt eine Speicheradresse als Parameter. Wie wir vom vorherigen Befehl wissen, entspricht der hexadezimale Wert \$FB dem dezimalen Wert 251, also einer 8bit-Adresse.

Nun gehen wir alle Zeilen für den Befehl LDX durch und sehen in der zweiten Spalte nach, ob wir für die Verwendungsart „Inhalt von 8bit Adresse laden“ einen Eintrag finden. In der zweiten Zeile werden wir fündig und aus der dritten Spalte können wir nun den passenden Befehlscode auslesen.

Der zu erzeugende Befehlscode lautet 166. Auf diesen folgt dann wiederum die Speicheradresse 251.

Der Befehl LDX \$FB wird also in die Zahlenfolge 166, 251 übersetzt.

Auf dem Zettel, welcher das bisherige Ergebnis der Übersetzung enthält, steht nun die Zahlenfolge 169, 0, 133, 251, 166, 251

INX

Der Befehl INX ist gültig, da er in der ersten Spalte der Tabelle zu finden ist. Und da dieser Befehl keine Parameter hat, gibt es auch nur einen Eintrag in der Tabelle und in der dritten Spalte steht der Befehlscode 232.

Auf unserem Zettel steht nun die Zahlenfolge 169, 0, 133, 251, 166, 251, 232

STX \$0400

Der Befehl STX ist gültig, da er in der ersten Spalte der Tabelle zu finden ist. Und da der darauffolgenden Zahl kein # vorangestellt ist, handelt es sich dabei um eine Speicheradresse. In diesem Fall ist es eine 16bit-Adresse, nämlich \$0400 (1024 dezimal).

Nun gehen wir alle Zeilen für den Befehl STX durch und sehen in der zweiten Spalte nach, ob wir für die Verwendungsart „Inhalt an 16bit Adresse schreiben“ einen Eintrag finden. In der fünften Zeile werden wir fündig und aus der dritten Spalte können wir nun den passenden Befehlscode 142 auslesen.

Darauf folgen das niedwertige und das höherwertige Byte der Adresse. Der Befehl STX \$0400 wird also in die Zahlenfolge 142, 0, 4 übersetzt.

Auf unserem Zettel steht nun die Zahlenfolge 169, 0, 133, 251, 166, 251, 232, 142, 0, 4

LDY \$FB

Der Befehl LDY ist gültig, da er in der ersten Spalte der Tabelle zu finden ist. Aufgrund des fehlenden Zeichens # wird die dem Befehl folgende Zahl als Speicheradresse 251 erkannt, welche eine 8bit Adresse darstellt.

Nun gehen wir alle Zeilen für den Befehl LDY durch und sehen in der zweiten Spalte nach, ob wir für die Verwendungsart „Inhalt aus 8bit Adresse laden“ einen Eintrag finden. In der dritten Zeile werden wir fündig und aus der dritten Spalte können wir nun den passenden Befehlscode 164 auslesen.

Der Befehl LDY \$FB wird also in die Zahlenfolge 164, 251 übersetzt.

Auf unserem Zettel steht nun die Zahlenfolge 169, 0, 133, 251, 166, 251, 232, 142, 0, 4, 164. 251

INY

Der Befehl INY ist gültig, da er in der ersten Spalte der Tabelle zu finden ist. Und da dieser Befehl keine Parameter hat, gibt es auch nur einen Eintrag in der Tabelle und in der dritten Spalte steht der Befehlscode 200.

Auf unserem Zettel steht nun die Zahlenfolge 169, 0, 133, 251, 166, 251, 232, 142, 0, 4, 164. 251, 200

INY

Der Befehl INY ist gültig, da er in der ersten Spalte der Tabelle zu finden ist. Und da dieser Befehl keine Parameter hat, gibt es auch nur einen Eintrag in der Tabelle und in der dritten Spalte steht der Befehlscode 200.

Auf unserem Zettel steht nun die Zahlenfolge 169, 0, 133, 251, 166, 251, 232, 142, 0, 4, 164. 251, 200, 200

STY \$0401

Der Befehl STY ist gültig, da er in der ersten Spalte der Tabelle zu finden ist. Wiederum fehlt das Zeichen # und daher wird die auf den Befehl folgende Speicheradresse erkannt. In diesem Fall ist es die 16bit-Adresse \$0401 (dezimal 1025).

Nun gehen wir alle Zeilen für den Befehl STY durch und sehen in der zweiten Spalte nach, ob wir für die Verwendungsart „Inhalt ab 16bit Adresse schreiben“ einen Eintrag finden. In der sechsten Zeile werden wir fündig und aus der dritten Spalte können wir nun den passenden Befehlscode 140 auslesen.

Darauf folgen das niedwertige und höherwertige Byte der Adresse. Der Befehl STY \$0401 wird also in die Zahlenfolge 140, 1, 4 übersetzt.

Auf unserem Zettel steht nun die Zahlenfolge 169, 0, 133, 251, 166, 251, 232, 142, 0, 4, 164. 251, 200, 200, 140, 1, 4

RTS

Der Befehl RTS ist gültig, da er in der ersten Spalte der Tabelle zu finden ist. Und da dieser Befehl keine Parameter hat, gibt es auch nur einen Eintrag in der Tabelle und in der dritten Spalte steht der Befehlscode 96.

Auf unserem Zettel steht nun die Zahlenfolge 169, 0, 133, 251, 166, 251, 232, 142, 0, 4, 164, 251, 200, 200, 140, 1, 4, 96

Wir haben nun unser Assembler-Programm vollständig in Maschinensprache übersetzt.

Nun ein paar Verständnisfragen:

1.) Hätten wir den Befehl LDA \$FB übersetzen können?

Nein, denn der Befehl LDA ist zwar gültig, weil er in der ersten Spalte der Tabelle vorkommt, aber es gibt keinen Tabelleneintrag für die Verwendungsart „Inhalt von 8bit Adresse laden“.

Wir müssten die Übersetzung also mit einem Fehler abbrechen.

2.) Hätten wir den Befehl XYZ übersetzen können?

Nein, denn unsere Tabelle enthält keinen Eintrag für einen Befehl namens XYZ und dieser wird daher nicht als gültiger Befehl erkannt.

Auch in diesem Fall müssten wir die Übersetzung mit einem Fehler abbrechen.

3.) Hätten wir den Befehl LDX #\$64 übersetzen können?

Nein, denn der Befehl LDX ist zwar gültig, weil er in der ersten Spalte der Tabelle vorkommt, aber es gibt keinen Tabelleneintrag für die Verwendungsart „Fixen Zahlenwert laden“.

Wiederum müssten wir die Übersetzung mit einem Fehler abbrechen.

Damit wir auch die Befehle LDA \$FB und LDX #\$64 übersetzen können, müssten wir unsere Tabelle um die grün markierten Einträge erweitern:

| Befehl | Verwendungsart | Zu erzeugender Befehlscode |
|--------|-----------------------------------|----------------------------|
| LDA | Fixen Zahlenwert laden | 169 |
| LDA | Inhalt von 8bit Adresse laden | 165 |
| LDX | Inhalt von 8bit Adresse laden | 166 |
| LDX | Fixen Zahlenwert laden | 162 |
| LDY | Inhalt von 8bit Adresse laden | 164 |
| STA | Inhalt an 8bit Adresse schreiben | 133 |
| STX | Inhalt an 16bit Adresse schreiben | 142 |

| Befehl | Verwendungsart | Zu erzeugender Befehlscode |
|--------|-----------------------------------|----------------------------|
| STY | Inhalt an 16bit Adresse schreiben | 140 |
| INX | | 232 |
| INY | | 200 |
| RTS | | 96 |

Nun wissen Sie also, wie so ein Übersetzungsprogramm arbeitet und Assembler-Befehle in Maschinensprache übersetzt.

Unsere Arbeit als Übersetzungs-Programm ist aber noch nicht abgeschlossen, denn wir müssen die Zahlenfolge, die auf dem Zettel steht noch in den Speicher schreiben, damit das Maschinenprogramm dann mit dem Befehl SYS gestartet werden kann.

Nehmen wir als Startadresse wiederum die Adresse 5376.

Die Zahlen, welche auf unserem Zettel stehen, können wir nun entweder über einzelne POKE-Befehle in den Speicher bekommen oder wir bauen uns den entsprechenden BASIC-Loader. Da dies den komfortableren Weg darstellt, entscheiden wir uns für diesen Weg.

```

10 FOR I=5376 TO 5393
20 READ A
30 POKE I,A
40 NEXT I
50 END
60 DATA 169, 0, 133, 251, 166, 251, 232, 142, 0, 4, 164, 251, 200, 200, 140, 1, 4, 96

```

Speichern Sie das Programm unter dem Namen „ASMLOADER“ auf der Diskette ab, damit Sie das Programm jederzeit wieder laden und ausführen können.

Nach dem Speichern starten Sie den BASIC-Loader mit RUN, damit das Maschinenprogramm in den Speicher geschrieben wird.

```
***** COMMODORE 64 BASIC V2 *****  
64K RAM SYSTEM 38911 BASIC BYTES FREE  
READY.  
10 FOR I=5376 TO 5393  
20 READ A  
30 POKE I,A  
40 NEXT I  
50 END  
60 DATA 169,0,133,251,166,251,232,142,0,  
4,164,251,200,200,140,1,4,96  
SAVE "ASMLoader",8  
SAVING ASMLoader  
READY.  
RUN  
READY.
```

So sieht also der Weg vom Assembler-Programm zum ausführbaren Maschinenprogramm aus.

Doch bevor wir das Programm starten, wollen wir uns ansehen, was das Programm eigentlich tut.

LDA #\$00
STA \$FB

LDX \$FB
INX
STX \$0400

LDY \$FB
INY
INY
STY \$0401

RTS

Im ersten Befehl LDA #\$00 wird der Akkumulator mit dem Wert 0 geladen. Im nächsten Befehl STA \$FB wird der Inhalt des Akkumulators an die Speicherstelle 251 geschrieben. Diese Speicherstelle enthält also nun den Wert 0.

Dann wird durch den nächsten Befehl LDX \$FB das X Register mit dem Inhalt der Speicherstelle 251 geladen, d.h. das X Register enthält nun den Wert 0.

Durch den Befehl INX wird der Inhalt des X Registers um 1 erhöht, d.h. es enthält nun den Wert 1. Dieser Wert wird dann durch den Befehl STX \$0400 an die Speicheradresse \$0400 (dezimal 1024) geschrieben.

Diese Speicheradresse kennen wir bereits aus dem vorherigen Kapitel, denn es ist die Adresse der linken, oberen Ecke im Bildschirmspeicher.

Folglich wird durch diesen Befehl ein A in die linke, obere Ecke des Bildschirms geschrieben.

Nun wird das Y Register durch den Befehl LDY \$FB mit dem Inhalt der Speicherstelle 251 geladen, also mit dem Wert 0.

Dann wird dessen Inhalt zweimal durch den Befehl INY um 1 erhöht, d.h. es enthält nun den Wert 2.

Durch den Befehl STY \$0401 wird dieser Wert dann an die Adresse \$0401 (dezimal 1025) geschrieben. Dies bewirkt, das rechts neben dem A ein B am Bildschirm erscheint.

Der Befehl RTS beendet das Programm und kehrt wieder zu BASIC zurück.

Nun werden wir das Programm starten, um zu überprüfen, ob es auch das gewünschte Ergebnis liefert.

Löschen Sie also den Bildschirm durch Drücken von CTRL + CLR/HOME, bewegen den Cursor um eine oder mehrere Zeilen nach unten und starten das Maschinenprogramm durch Eingabe von SYS 5376. Wenn alles gut gegangen ist, sollten Sie das soeben beschriebene Ergebnis erhalten.



Ich hoffe, dass ich Ihnen nun die Unterschiede zwischen Maschinensprache und Assembler näherbringen konnte.

Die Assembler-Sprache ist im Grunde eine für uns Menschen besser lesbare Form der Maschinensprache.

Im nächsten Kapitel geht es nun wirklich los mit der Assembler-Programmierung :)

2.2 Einführung in die Arbeit mit SMON

2.2.1 Was ist SMON?

Das Programm SMON brauchen wir nun in allererster Linie, um Programme in Assembler schreiben und ausführen zu können.

Es übersetzt die Assembler-Befehle, die ja als kurze Textbefehle eingegeben werden, in die Befehlscodes, welche die CPU dann verarbeiten kann.

Diesen Übersetzungs-Vorgang nennt man Assemblieren.

Der SMON bietet aber auch den umgekehrten Weg an, nämlich das Rückverwandeln dieser Zahlenfolgen in ein Assembler-Programm. Diesen Vorgang nennt man Disassemblieren.

Darüber hinaus hat man jedoch noch viele weitere Möglichkeiten, vom Speichern / Laden unserer Assembler-Programme auf / von Diskette oder Kassette angefangen bis hin zum Betrachten der Inhalte des Arbeitsspeichers, um nur einige Möglichkeiten zu nennen.

Auf der Webseite <https://www.c64-wiki.de/wiki/SMON> ist eine sehr umfassende Beschreibung des SMON zu finden die Sie sicher immer wieder mal gut gebrauchen können.

SMON ist übrigens selbst in Maschinensprache geschrieben, deswegen müssen wir ihn, nachdem wir ihn von Diskette geladen haben, mit SYS 49152 starten und nicht mit RUN.

Was es mit der Adresse 49152 auf sich hat, ist momentan nicht so wichtig, ich komme später noch darauf zurück.

Merken Sie sich im Moment nur, dass der SMON nach dem Laden durch den LOAD-Befehl ab Adresse 49152 im Speicher liegt und daher mit SYS 49152 gestartet werden muss und nicht mit RUN, wie es bei BASIC-Programmen der Fall ist.

2.2.2 Laden und Starten von SMON

Wir laden den SMON, wie auf folgendem Screenshot zu sehen, und starten ihn durch SYS 49152. Der Befehl NEW, der vor dem SYS Befehl ausgeführt wird, soll laut den Angaben auf der Seite <https://www.c64-wiki.de/wiki/SMON> die Fehlermeldung OUT OF MEMORY verhindern, wenn man auch BASIC benutzen will.

```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
LOAD"SMONPC000",8,1
SEARCHING FOR SMONPC000
LOADING
READY.
NEW
READY.
SYS 49152

PC  SR  AC  XR  YR  SP  MU-BDIZC
;C00B B0 C2 00 00 F6  10110000
.
```

Was wir hier sehen sind die Inhalte der CPU-Register in hexadezimaler Form.

Die hexadezimale Schreibweise werde ich ab jetzt auch beibehalten, nicht zuletzt deswegen, weil auch der SMON alle Werte in dieser Form anzeigt.

Wie können wir nun Assembler-Befehle eingeben?

Zunächst müssen wir uns überlegen, ab welcher Adresse wir unsere Befehle in den Arbeitsspeicher schreiben wollen. Bleiben wir bei der bisher verwendeten Adresse 5376. Wir brauchen nun aber die hexadezimale Darstellung, da der SMON Zahlen in dieser Form entgegennimmt bzw. ausgibt.

Hier kann uns der SMON bereits behilflich sein, denn wenn wir Das Zeichen # gefolgt von einer dezimalen Zahl eingeben liefert uns SMON die entsprechende hexadezimale Darstellung.

```
PC  SR  AC  XR  YR  SP  MU-BDIZC
;C00B B0 C2 00 00 F6  10110000
.#5376.
```

Liefert uns das Ergebnis:

```
PC  SR  AC  XR  YR  SP  NU-BDIZC  
;C00B  B8  C2  00  00  F6  10110000  
1500  5376
```

Die hexadezimale Darstellung der dezimalen Zahl 5376 lautet also \$1500.

Wenn wir umgekehrt eine hexadezimale Zahl in eine Dezimalzahl umwandeln wollen, dann funktioniert das durch Eingabe des Zeichens \$ gefolgt von der hexadezimalen Zahl.

```
PC  SR  AC  XR  YR  SP  NU-BDIZC  
;C00B  B8  C2  00  00  F6  10110000  
1500  5376  
. $1500■
```

Liefert uns das Ergebnis:

```
PC  SR  AC  XR  YR  SP  NU-BDIZC  
;C00B  B8  C2  00  00  F6  10110000  
1500  5376  
1500  5376  
.■
```

Die untere Zeile ist das Resultat unserer letzten Anweisung, doch dieses mal ist der zweite Wert 5376, an dem wir interessiert sind, denn wir wollten ja die dezimale Darstellung der hexadezimalen Zahl \$1500 ermitteln.

2.2.3 Eingabe von Assembler-Befehlen

Damit wir nun mit der Eingabe ab Adresse \$1500 beginnen können, müssen wir dem SMON dem Befehl A gefolgt von der Adresse \$1500 geben.

Das A steht für Assemble und zeigt dem SMON an, dass wir Assembler-Befehle eingeben wollen.

Geben wir also folgendes ein:

```
PC  SR  AC  XR  YR  SP  NU-BDIZC  
;C00B  B8  C2  00  00  F6  10110000  
1500  5376  
1500  5376  
.A 1500■
```

SMON meldet sich mit folgender Anzeige:

```
PC  SR  AC  XR  YR  SP  NU-BDIZC
;C00B  B0  C2  00  00  F6  10110000
1500  5376
1500  5376
.A 1500
1500 ■
```

Nun können wir unseren ersten Assembler-Befehl eingeben. Da wir im vorherigen Kapitel ja bereits auf dem Papier ein Assembler-Programm geschrieben haben, können wir dieses nun verwenden, um uns mit den Funktionen des SMON vertraut zu machen.

Der erste Befehl in unserem Programm ist der Befehl LDA #\$00 und daher geben wir diesen nun ein.

```
PC  SR  AC  XR  YR  SP  NU-BDIZC
;C00B  B0  C2  00  00  F6  10110000
.A 1500
1500 LDA #$00■
```

Wir bekommen folgende Rückmeldung:

```
PC  SR  AC  XR  YR  SP  NU-BDIZC
;C00B  B0  C2  00  00  F6  10110000
.A 1500
1500 A9  00      LDA #00
1502 ■
```

Was ist passiert? SMON hat unseren Befehl LDA #\$00 entgegengenommen und verarbeitet.

Wie ist er dabei vorgegangen? Genau so, wie wir es im vorherigen Kapitel getan haben, als wir selbst in die Rolle des Übersetzers geschlüpft sind.

Er hat zunächst die Zeichenfolge LDA als gültigen Assembler-Befehl erkannt.

Durch das folgende Zeichen # hat er erkannt, dass wir mit dem nachfolgenden Wert einen Zahlenwert und keine Speicheradresse meinen.

Das Zeichen \$ zeigt an, dass eine Zahl in hexadezimaler Form folgt.

\$00 steht hier für die hexadezimale Form der dezimalen Zahl 0.

Damit hat der SMON alle Informationen, die er braucht, um den Befehl in Maschinensprache zu übersetzen.

Links neben dem Befehl LDA ist das Ergebnis der Übersetzung zu sehen.

In der Speicherstelle \$1500 steht nun der Wert \$A9 und in der Speicherstelle \$1501 der Wert \$00, der Befehl beansprucht also zwei Bytes.

\$A9 ist die hexadezimale Form des dezimalen Befehlscodes 169 und \$00 wie gesagt die hexadezimale Darstellung des dezimalen Wertes 0.

Der nächste Befehl würde nun an die Adresse \$1502 kommen, wie man an auch in obigem Screenshot sieht (Adresse links neben dem Cursor)

Geben wir nun den nächsten Befehl STA \$FB ein. Dieses mal lassen wir das Zeichen # hinter dem Befehl LDA weg, um SMON anzuzeigen, dass wir nun keinen Zahlenwert, sondern eine Speicheradresse meinen.

```
PC  SR  AC  XR  YR  SP  NU-BDIZC
;C00B  B8  C2  00  00 F6  10110000
.A 1500
1500 A9 00      LDA #00
1502 STA $FB■
```

Wir erhalten folgendes Ergebnis:

```
PC  SR  AC  XR  YR  SP  NU-BDIZC
;C00B  B8  C2  00  00 F6  10110000
.A 1500
1500 A9 00      LDA #00
1502 85 FB      STA   FB
1504 ■
```

Geben Sie nun nacheinander auf die selbe Art und Weise die restlichen Befehle ein.

Nachdem Sie den letzten Befehl RTS eingegeben haben, geben Sie ein F ein. Dadurch zeigen wir SMON an, dass wir den Eingabemodus verlassen und wieder in den Befehlsmodus wechseln wollen.

```
PC  SR  AC  XR  YR  SP  NU-BDIZC
;C00B  B8  C2  00  00 F6  10110000
.A 1500
1500 A9 00      LDA #00
1502 85 FB      STA   FB
1504 A6 FB      LDX   FB
1506 E8          INX
1507 8E 00 04    STX  0400
150A A4 FB      LDY   FB
150C C8          INY
150D C8          INY
150E 8C 01 04    STY  0401
1511 60          RTS
1512 F■
```

Nun drücken Sie die RETURN-Taste und Sie erhalten folgendes Ergebnis:

```
;C00B B0 C2 00 00 F6 10110000
.A 1500
1500 A9 00 LDA #00
1502 85 FB STA FB
1504 A6 FB LDX FB
1506 E8 INX
1507 BE 00 04 STX 0400
150A A4 FB LDY FB
150C C8 INY
150D C8 INY
150E 8C 01 04 STY 0401
1511 60 RTS
1512 F
,1500 A9 00 LDA #00
,1502 85 FB STA FB
,1504 A6 FB LDX FB
,1506 E8 INX
,1507 BE 00 04 STX 0400
,150A A4 FB LDY FB
,150C C8 INY
,150D C8 INY
,150E 8C 01 04 STY 0401
,1511 60 RTS
-----.
.
```

SMON zeigt uns das eingegebene Programm nochmals an. Die Kommas vor den Zeilen bedeuten, dass man in dieser Zeile Änderungen vornehmen kann.

Was sehen wir nun auf diesem Bild?

Wir sehen, dass unser Programm an der Adresse \$1500 mit dem Befehl LDA #00 beginnt und an der Adresse \$1511 mit dem Befehl RTS endet.

Unser Programm belegt also 18 Bytes im Speicher.

Links neben jedem Assembler-Befehl wird in hexadezimaler Form angezeigt, in welche Zahlenfolge dieser Befehl übersetzt wurde.

Hier sieht man auch deutlich, wie viele Bytes die einzelnen Befehle im Speicher belegen. Die ersten drei Befehle belegen zwei Bytes, dann folgt mit dem Befehl INX ein Befehl, welcher nur ein Byte belegt. Als nächstes sieht man den Befehl STX 0400, welcher drei Bytes belegt usw.

Sehen Sie den Unterschied zwischen dem Befehl STA \$FB und dem Befehl STX \$0400?

Der Befehl STA \$FB belegt nur zwei Bytes im Speicher (ein Byte für den Befehlscode und ein Byte für die 8bit-Speicheradresse).

Der Befehl STX \$0400 belegt hingegen drei Bytes im Speicher (ein Byte für den Befehlscode und zwei weitere Bytes, um sowohl das niederwertige als auch das höherwertige Byte der 16bit-Speicheradresse \$0400 darzustellen)

2.2.4 Starten eines Assembler-Programms

Wir starten unser Programm durch den SMON-Befehl G gefolgt von der Startadresse unseres Programms. In unserem Fall würde der Befehl zum Starten des Programms also G 1500 lauten.

Doch bevor wir diesen Befehl eingeben, löschen wir zunächst den Bildschirm durch Drücken von SHIFT + CLR/HOME.

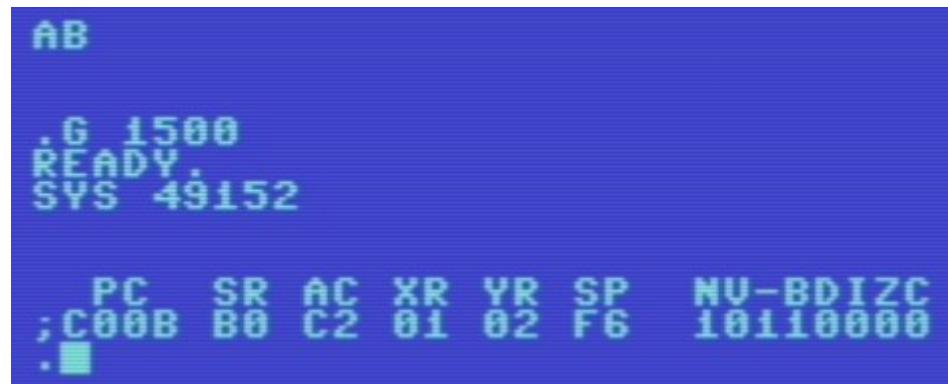
Dann bewegen wir den Cursor um eine oder mehrere Zeilen nach unten, geben einen Punkt gefolgt von dem Befehl G 1500 ein und drücken die RETURN-Taste.

Unser Programm wird gestartet und wir landen durch den Befehl RTS am Ende wieder im BASIC.



```
AB
.G 1500
READY.
```

Doch wie kommen wir nun wieder zurück zu SMON? Auf dem gleichen Weg wie zu Beginn, nämlich durch Eingabe von SYS 49152.



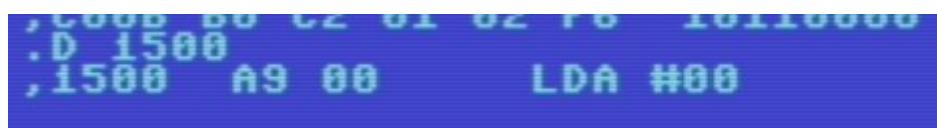
```
AB
.G 1500
READY.
SYS 49152
PC C00B SR B0 AC C2 XR 01 YR 02 SP F6 MU-BDIZC 10110000
```

Damit wir beim nächsten Start unseres Programms am Ende nicht im BASIC landen, sondern im SMON verbleiben, müssen wir eine kleine Änderung vornehmen. Wir müssen den Befehl RTS durch den Befehl BRK ersetzen.

Doch wie können wir unser Programm nun wieder anzeigen lassen? Dazu verwenden wir die Möglichkeit des Disassemblierens. Durch das Disassemblieren können wir aus Zahlenwerten im Speicher wieder ein für uns lesbaren Assembler-Programm anzeigen lassen.

Geben Sie also den Befehl D 1500 ein und drücken die RETURN-Taste.

Der erste Befehl unseres Programms wird angezeigt.



```
D 1500
;1500 A9 00      LDA #00
```

Durch mehrmaliges Drücken der Leertaste wird unser Programm wieder Befehl für Befehl angezeigt.

Wenn die gestrichelte Linie angezeigt wird, drücken Sie die ESC-Taste, um wieder in den Befehlsmodus zu gelangen.

The screenshot shows a memory dump and assembly code. The memory dump at the top shows addresses from 1500 to 1511 with their corresponding hex values. Below it is the assembly code:

```
;1500 D0 02 01 02 F6 10110000
.D 1500
;1500 A9 00      LDA #00
;1502 85 FB      STA FB
;1504 A6 FB      LDX FB
;1506 E8         INX
;1507 8E 00 04    STX 0400
;150A A4 FB      LDY FB
;150C C8         INV
;150D C8         INV
;150E 8C 01 04    STY 0401
;1511 60         RTS
```

A dashed line separates the memory dump from the assembly code. At the bottom, there are two small square icons.

Nun bewegen wir den Cursor auf den ersten Buchstaben des Befehls RTS.

The assembly code is shown again, but the cursor is now positioned under the first character of the RTS instruction at address 1511.

```
;1500 D0 02 01 02 F6 10110000
;150E 8C 01 04   STY 0401
;1511 60         RTS
```

A dashed line separates the memory dump from the assembly code. At the bottom, there is one small square icon.

Nun geben wir den Befehl BRK ein und drücken die RETURN-Taste.

The assembly code is shown again, but the cursor is now positioned under the first character of the BRK instruction at address 1511.

```
;1500 D0 02 01 02 F6 10110000
;1511 00         BRK
```

A dashed line separates the memory dump from the assembly code. At the bottom, there is one small square icon.

Wie wir sehen hat sich links der Befehlscode auf den Befehl BRK angepasst, denn dieser hat den Befehlscode 0.

Löschen Sie nun wieder den Bildschirm, bewegen den Cursor um eine oder mehrere Zeilen nach unten und geben wieder einen Punkt gefolgt durch den Befehl G 1500 ein.

The assembly code is shown again, but the cursor is now positioned under the first character of the G 1500 command at address 1512.

```
AB
.G 1500
PC  SR  AC  XR  YR  SP  MU-BDIZC
;1512 30 00 01 02 F6 00110000
.■
```

A dashed line separates the memory dump from the assembly code. At the bottom, there is one small square icon.

Wie vorhin wird unser Programm gestartet, doch nun landen wir nicht wieder im BASIC, sondern verbleiben im SMON.

Was Sie hier im unteren Bereich sehen, sind die aktuellen Registerinhalte (PC = Program Counter, SR = Statusregister, AC = Akkumulator, XR = X Register, YR = Y Register und SP = Stack Pointer)

Dahinter werden die einzelnen Flags im Statusregister angezeigt – doch dazu kommen wir später. Mit dem Befehl R können Sie sich übrigens jederzeit die aktuellen Registerinhalte anzeigen lassen.

```
AB
.G 1500
; PC  SR  AC  XR  YR  SP  NU-BDIZC
;1512 30 00 01 02 F6  00110000
.R
; PC  SR  AC  XR  YR  SP  NU-BDIZC
;1512 30 00 01 02 F6  00110000
.■
```

Sie können die Registerinhalte auch ändern. Wie man hier sieht, enthält der Akkumulator aktuell den Wert 0. Nun wollen wir den Inhalt des Akkumulators auf den Wert 5 ändern.

Dazu bewegen Sie den Cursor auf die erste der beiden Ziffern unter dem Text AC und geben den neuen Wert 05 ein.

```
;1512 30 00 01 02 F6  00110000
.R
; PC  SR  AC  XR  YR  SP  NU-BDIZC
;1512 30 05■01 02 F6  00110000
.
```

Nach dem Drücken der RETURN-Taste landet der Cursor wieder hinter dem Punkt und wenn Sie nun zur Kontrolle ihrer Änderung wieder den Befehl R eingeben, erhalten Sie folgende Anzeige:

```
; PC  SR  AC  XR  YR  SP  NU-BDIZC
;1512 30 05 01 02 F6  00110000
.R
; PC  SR  AC  XR  YR  SP  NU-BDIZC
;1512 30 05 01 02 F6  00110000
.■
```

Wie man hier sieht, enthält der Akkumulators nun den Wert 5.

2.2.5 Anzeigen des Inhalts von Speicherbereichen

Ein weiterer hilfreicher Befehl ist M Startadresse Endadresse

Damit kann man sich den Inhalt des Arbeitsspeichers beginnend bei der Startadresse und endend bei der Endadresse anzeigen lassen.

Unser Programm beginnt wie gesagt im Arbeitsspeicher bei Adresse \$1500 und endet an der Adresse \$1511. Sehen wir uns doch mal an, wie unser Programm im Speicher aussieht.

Wichtig:

SMON verlangt bei der Angabe der Endadresse jedoch den um 1 erhöhten Wert, d.h. wir müssen hier statt \$1511 den Wert \$1512 angeben. Es gibt auch noch andere SMON-Befehle, bei denen man dies beachten muss.

Geben wir also den Befehl M 1500 1512 ein, drücken die RETURN-Taste und erhalten folgendes Ergebnis:

| PC | SR | AC | XR | YR | SP | NU-BDIZC |
|--------|------|----|----|----|----|-------------|
| :1512 | 30 | 05 | 01 | 02 | F6 | 00110000 |
| M 1500 | 1512 | | | | | |
| :1500 | A9 | 00 | 85 | FB | A6 | FB E8 8E |
| :1508 | 00 | 04 | A4 | FB | C8 | C8 8C 01 |
| :1510 | 04 | 00 | FF | FF | FF | FF 00 00 |
| . | | | | | | II .. |

Wie wir bereits wissen, zeigt SMON sämtliche Zahlenwerte in hexadezimaler Form an und deswegen sehen wir den Maschinencode unseres Programms nun ebenfalls in hexadezimaler Form.

Das erste Byte unseres Programm steht an Adresse \$1500. Hier sieht man den Befehlscode des Befehls LDA (\$A9) gefolgt von dem Wert \$00.

Wir erinnern uns: Der erste Befehl in unserem Programm lautet LDA #\$00

Darauf folgen die weiteren Befehle mit ihren Parametern und das letzte Byte unseres Programms steht schließlich an Adresse \$1511. Neben dem Wert \$04 an der Adresse \$1510 sieht man den Wert \$00.

Dies ist der Befehlscode des Befehls BRK. Die restlichen Bytes in dieser Zeile (FF, ...) gehören nicht mehr zu unserem Programm.

Schreiben wir nun ein Programm, welches ein A in Zeile 12 und Spalte 20 schreibt.

Wie wir bereits wissen, beginnt der Bildschirmspeicher bei Adresse \$0400 (dezimal 1024).

Der Bildschirm des C64 zeigt 25 Zeilen zu je 40 Spalten an, d.h. die Adresse der genannten Position errechnet sich durch $1024 + (11 \text{ Zeilen} * 40 \text{ Spalten} + \text{Spalte } 20) - 1 = 1483$ oder hexadezimal \$05CB.

Wir löschen mal den Bildschirm mit der Tastenkombination SHIFT + CLR/HOME und geben wie zuvor A 1500 ein:

```
.A 1500■
```

Geben Sie nach Drücken der RETURN Taste die folgenden drei Assembler-Befehle ein und verlassen Sie den Eingabemodus wieder mit dem Befehl F, damit Sie wieder in den Befehlsmodus zurückgelangen.

```
.A 1500
1500 A9 01 LDA #01
1502 8D CB 05 STA 05CB
1505 00 BRK
1506 F
;1500 A9 01 LDA #01
;1502 8D CB 05 STA 05CB
;1505 00 BRK
-----■
```

Löschen Sie wiederum den Bildschirm mit der Tastenkombination SHIFT + CLR/HOME

Nun wollen wir unser Programm starten.

Durch das Löschen des Bildschirms ist jedoch der Punkt verschwunden, welche jede SMON-Anweisung einleitet.

Daher geben wir einen Punkt ein und gleich dahinter die Anweisung G 1500:

```
.G 1500■
```

Wie erwartet wird in Zeile 12 / Spalte 20 der Buchstabe „A“ angezeigt:

```
.G 1500
PC SR AC XR YR SP NU-BDIZC
;1506 30 01 00 00 F6 00110000
.■
```

A

Nun wollen wir gerne statt einem A ein B anzeigen.

Wir müssen unser Programm also ändern, doch da wir den Bildschirm gelöscht haben, sehen wir unser Programm nicht mehr.

Kein Problem, wir werden unser Programm durch Eingabe von D 1500 wieder sichtbar machen.

2.2.6 Disassemblieren – aus Maschinensprache wird wieder Assembler

Den Befehl D haben wir bereits vorhin kennengelernt, als wir von dem Wechsel von BASIC zu SMON unser Programm wieder anzeigen wollten, um den Befehl RTS durch den Befehl BRK ersetzen zu können.

Durch den Befehl D wird der Speicherinhalt ab der angegebenen Adresse wieder zurück in ein Assembler-Programm verwandelt. Man nennt diesen Vorgang auch Disassemblieren, also das Gegenteil zum Vorgang des Assemblierens.

Beim Assemblieren werden Assembler-Befehle in Maschinensprache übersetzt, d.h. in eine Folge von Zahlenwerten. Beim Disassemblieren wird aus diesen Zahlenwerten dann wieder ein für uns lesbares Assembler-Programm.

Und da unser Programm ja ab Adresse \$1500 im Speicher steht, mussten wir diese Adresse auch als Parameter angeben.

Bei der Ausführung des Befehls wird zunächst die erste Zeile unseres Programms angezeigt und durch Drücken der Leertaste kommt eine Zeile nach der anderen wieder zum Vorschein.

Sobald die gestrichelte Linie nach dem BRK Befehl angezeigt wird, drücken Sie die Escape-Taste und Sie landen wieder im Befehlsmodus des SMON.



```
.G 1500
PC  SR  AC  XR  YR  SP    MU-BDIZC
;1506 30 01 00 00 F6  00110000
;D 1500
;1500 A9 01      LDA #01
;1502 8D CB 05    STA 05CB
;1505 00          BRK
-----
```

Nun bewegen wir den Cursor rechts neben den Befehl LDA und ändern den Wert \$01 in den Wert \$02:

```
.G 1500
    PC  SR  AC  XR  YR  SP  NU-BDIZC
;1506 30 01 00 00 F6  00110000
.D 1500
;1500  A9  01      LDA #02■
;1502  8D  CB  05   STA 05CB
;1505  00          BRK
-----
.
A
```

Nach dem Drücken der RETURN Taste sieht man, dass sich der Wert rechts neben dem Befehlscode \$A9 auf den Wert \$02 verändert hat.

```
.G 1500
    PC  SR  AC  XR  YR  SP  NU-BDIZC
;1506 30 01 00 00 F6  00110000
.D 1500
;1500  A9  02      LDA #02
;1502  8D  CB  05   STA 05CB
;1505  00          BRK
-----
.
A
```

Nun bewegen wir den Cursor nach oben hinter den Befehl G 1500 und drücken die RETURN Taste, wodurch unser Programm erneut gestartet wird. Wir sehen die Auswirkung der Änderung, nun wird anstelle des A ein B angezeigt.

```
.G 1500
    PC  SR  AC  XR  YR  SP  NU-BDIZC
;1506 30 02 00 00 F6  00110000
.D 1500
;1500  A9  02      LDA #02
;1502  8D  CB  05   STA 05CB
;1505  00          BRK
-----
.
B
```

Experimentieren Sie nun ein wenig, ändern Sie beispielsweise den Wert \$02 in \$03 oder verändern Sie die Adresse. Starten Sie das Programm erneut und beobachten Sie die Auswirkung Ihrer Änderungen.

Ach ja, falls Sie SMON wieder verlassen und zu BASIC zurückkehren wollen, funktioniert das mit dem Befehl X.

Zum besseren Verständnis möchte ich Ihnen noch ein weiteres Beispiel zum Thema Disassemblieren vorstellen, nur dieses mal beginnen wir in BASIC, also auf der anderen Seite.

Geben Sie folgendes BASIC-Programm ein und starten es mit RUN, damit das Maschinenprogramm, welches durch die Zahlencodes repräsentiert wird, durch die POKE Befehle in den Speicher gelangt.

Wir nehmen dieses mal zur Abwechslung mal eine andere Adresse, nämlich die 12288 oder \$3000.

The screenshot shows a Commodore 64 screen with a black background and white text. At the top, it displays "**** COMMODORE 64 BASIC V2 ****" and "64K RAM SYSTEM 38911 BASIC BYTES FREE". Below this, the word "READY." is printed twice. A series of BASIC commands are listed, starting with "10 POKE 12288,169" and ending with "110 POKE 12298,0". After these commands, the word "RUN" is typed. Finally, the word "READY." appears again at the bottom. The entire sequence of text is contained within a rectangular border.

```
**** COMMODORE 64 BASIC V2 ****  
64K RAM SYSTEM 38911 BASIC BYTES FREE  
READY.  
10 POKE 12288,169  
20 POKE 12289,200  
30 POKE 12290,162  
40 POKE 12291,85  
50 POKE 12292,160  
60 POKE 12293,202  
70 POKE 12294,105  
80 POKE 12295,16  
90 POKE 12296,232  
100 POKE 12297,200  
110 POKE 12298,0  
RUN  
READY.
```

Unser Programm steht nun von Adresse 12288 (\$3000) bis zur Adresse 12298 (\$300A) im Arbeitsspeicher.

Das werden wir nun mit SMON überprüfen!

Wir laden den SMON mit LOAD „SMONPC000“,8,1 in den Speicher und starten ihn mit SYS 49152.

Sobald der SMON gestartet ist, geben Sie den Befehl D 3000 wie auf dem folgenden Screenshot zu sehen ein.

```
PC  SR AC XR YR SP MU-BDIZC
;C00B B8 C2 00 00 F6 10110000
.D 3000■
```

Wir weisen den SMON dadurch an, die Zahlencodes, die ab der Adresse \$3000 im Speicher liegen, wieder in ein Assembler-Programm zu verwandeln.

Wir erhalten folgendes Ergebnis:

```
.D 3000
,3000  A9 C8    LDA #C8
,3002  A2 55    LDX #55
,3004  A0 CA    LDY #CA
,3006  69 10    ADC #10
,3008  E8      INX
,3009  C8      INY
,300A  00      BRK
-----■
```

SMON hat uns also das Maschinenprogramm, das wir vorhin mit dem BASIC-Programm Byte für Byte in den Speicher geschrieben haben, durch das Disassemblieren wieder als Assembler-Programm dargestellt.

Aus der nur für die CPU verständlichen Folge von Zahlen wird also durch das Disassemblieren für uns Menschen lesbarer Code in Assembler-Sprache.

Starten wir das Programm einfach mal mit dem Befehl G 3000

```
.G 3000
PC  SR AC XR YR SP MU-BDIZC
;300B B8 D8 56 CB F6 10110000
.■
```

Überprüfen wir nun ob die Inhalte des Akkumulators, des X Registers und des Y Registers stimmen.

Durch den ersten Befehl wird der Akkumulator mit dem Wert \$C8 (dezimal 200), das X-Register mit dem Wert \$55 (dezimal 85) und das Y Register mit dem Wert \$CA (dezimal 202) geladen.

Als nächstes wird der Wert \$10 (dezimal 16) zum Inhalt des Akkumulators addiert, d.h. der Akkumulator enthält nun den Wert $200 + 16 = 216$, also den hexadezimalen Wert \$D8, was sich mit der obigen Anzeige für AC deckt.

Der Befehl INX erhöht den Inhalt des X Registers um 1, d.h. der neue Inhalt lautet $85 + 1 = 86$ bzw. hexadezimal \$56. Auch dieser Wert deckt sich mit der obigen Anzeige für XR.

Der Befehl INY erhöht den Inhalt des Y Registers um 1, d.h. der neue Inhalt lautet $202 + 1 = 203$ bzw. hexadezimal \$CB was sich ebenfalls mit der obigen Anzeige für YR deckt.

Passt also alles!

2.2.7 Speichern eines Assembler-Programms auf Diskette

Als Nächstes möchte Ihnen zeigen, wie Sie ein Assembler-Programm, das Sie eingegeben haben auf Diskette speichern und auch von dort wieder laden können.

Starten Sie also den SMON und geben Sie folgendes kurzes Programm ein:

```
READY.  
SYS 49152  
  
PC SR AC XR YR SP NU-BDIZC  
;C00B B0 C2 00 00 F6 10110000  
.A 3000  
3000 A9 C8 LDA #C8  
3002 A2 FF LDX #FF  
3004 A0 AC LDY #AC  
3006 69 37 ADC #37  
3008 E8 IMX  
3009 C8 IMY  
300A 00 BRK  
300B F  
.3000 A9 C8 LDA #C8  
.3002 A2 FF LDX #FF  
.3004 A0 AC LDY #AC  
.3006 69 37 ADC #37  
.3008 E8 IMX  
.3009 C8 IMY  
.300A 00 BRK  
.  
■
```

Ich habe hier einfach nur ein kurzes Programm mit ein paar Befehlen geschrieben, damit wir ein Programm haben, dass wir auf Diskette speichern und anschließend wieder von dort laden können.

Zum Speichern dient der Befehl

S „Name des Programms“ Startadresse Endadresse + 1

Wir geben unserem Programm den Namen TEST und speichern es mit folgendem Befehlen

S „TEST“ 3000 300B

```
-----  
.S"TEST" 3000 300B  
SAVING TEST  
.■
```

Geben Sie kein Leerzeichen zwischen dem Befehl S und dem Namen des Programms ein.

Es wird die Meldung „SAVING TEST“ angezeigt und das Programm wird unter diesem Namen auf der Diskette gespeichert.

2.2.8 Laden eines Assembler-Programms von Diskette

Nun starten wir den C64 neu, laden den SMON erneut und lassen uns den Speicherbereich ab \$3000 disassemblieren. Drücken Sie einfach ein paar mal die Leertaste um einige Zeilen anzuzeigen und anschließend die Escape-Taste um wieder zur Befehlseingabe des SMON zurückzukehren.

The screenshot shows the Commodore 64 screen with the SMON monitor running. The top line displays "READY." followed by "SYS 49152". Below this, a memory dump is shown starting at address \$C00B. The columns represent PC, SR, AC, XR, YR, SP, and MU-BDIZC. The dump shows several BRK (Break) instructions at addresses \$3000 through \$3009. Addresses \$3002 through \$3006 have FF values in the AC column, while others have 00. The MU-BDIZC column shows a mix of binary values and asterisks (***) for non-printable characters. A small square icon is visible in the bottom left corner of the screen area.

| PC | SR | AC | XR | YR | SP | MU-BDIZC |
|-------|------|----|----|----|----|----------|
| ,C00B | B8 | C2 | 00 | 00 | F6 | 10110000 |
| .D | 3000 | | | | | |
| ,3000 | 00 | | | | | BRK |
| | | | | | | |
| ,3001 | 00 | | | | | BRK |
| | | | | | | |
| ,3002 | FF | | | | | *** |
| ,3003 | FF | | | | | *** |
| ,3004 | FF | | | | | *** |
| ,3005 | FF | | | | | *** |
| ,3006 | 00 | | | | | BRK |
| | | | | | | |
| ,3007 | 00 | | | | | BRK |
| | | | | | | |
| ,3008 | 00 | | | | | BRK |
| | | | | | | |
| ,3009 | 00 | | | | | BRK |
| | | | | | | |
| . | ■ | | | | | |

Ich habe hier obige Ausgabe erhalten, diese kann bei Ihnen unterschiedlich sein. Sie zeigt den Inhalt des Speichers ab Adresse \$3000 VOR dem Laden unseres Programms von Diskette.

Nun laden wir das Programm, welches wir vorhin unter dem Namen TEST abgespeichert haben, wieder von der Diskette in den Arbeitsspeicher.

Dies funktioniert mit dem Befehl L“Name des Programms“, in unserem Fall also L“TEST“



Nun befindet sich unser Programm wieder im Arbeitsspeicher. Aber wo?

Das Programm wird beim Laden wieder ab Adresse \$3000 in den Arbeitsspeicher geschrieben, weil die Startadresse, welche beim Speichern angegeben wurde, mit in die Datei übernommen wurde.

Das ist auch der Grund, warum wir beim Ladebefehl keine Startadresse angeben mussten, weil diese bereits in der gespeicherten Datei vermerkt ist.

Nun wollen wir überprüfen, ob das Laden des Programms geklappt hat. Wir führen also den Befehl D 3000 aus und wie wir hier sehen, steht unser Programm tatsächlich wieder ab Adresse \$3000 im Speicher.

```
L"TEST"
SEARCHING FOR TEST
LOADING
.D 3000
,3000 A9 C8      LDA #C8
,3002 A2 FF      LDX #FF
,3004 A0 AC      LDY #AC
,3006 69 37      ADC #37
,3008 E8         INX
,3009 C8         INY
,300A 00         BRK
-----
,300B FF        ***
.■
```

Nun wissen wir also auch, wie wir Assembler-Programme auf Diskette speichern und wieder von dort in den Arbeitsspeicher laden.

Zum Thema Disassemblieren möchte ich noch folgendes anmerken:

Jeder Befehlscode der CPU ist einer Zahl zwischen 0 und 255 zugeordnet. Aber umgekehrt sind nicht alle diese Werte einem Befehlscode zugeordnet, d.h. es gibt durchaus Werte in diesem Bereich, zu denen es keinen entsprechenden Befehl gibt.

Trifft der SMON nun beim Disassemblieren auf einen Befehlscode, den er nicht in einen entsprechenden Assembler-Befehl übersetzen kann, wird dies durch *** angezeigt, wie man oben am Beispiel des Wertes \$FF (dezimal 255) sieht.

2.2.9 Automatisches Erstellen von DATA-Zeilen für einen BASIC-Loader

Ein cooles Feature des SMON möchte ich Ihnen noch zeigen.

Im ersten Kapitel haben wir gelernt, wie wir ein Maschinenprogramm mittels eines sogenannten BASIC-Loaders in den Arbeitsspeicher schreiben können. Die Bytes des Maschinenprogramms werden einfach in DATA-Zeilen untergebracht und nacheinander Byte für Byte in den Arbeitsspeicher geschrieben.

Angenommen, wir haben mit SMON ein Assembler-Programm geschrieben und wollen dieses in Form eines BASIC-Loaders zur Verfügung stellen. Durch den BASIC-Loader werden wir unabhängig von SMON und können unser Programm jederzeit auch ohne diesen laden und ausführen.

Die Maschinencodes unseres Assembler-Programms liegen im Hauptspeicher direkt vor uns, aber wie erstellen wir aus diesen Maschinencodes eine Reihe von DATA-Zeilen, die wir dann in unserem BASIC-Loader verwenden können?

SMON bietet uns hier genau jene Funktion, die wir brauchen.

Der Befehl lautet B Startadresse Endadresse+1

Mit diesem Befehl generiert uns SMON die DATA-Zeilen für unseren BASIC-Loader, welche die Maschinencodes unseres Assembler-Programms enthalten. Alles was wir dann noch ergänzen müssen, ist die FOR-Schleife, in der dann die Bytes aus den DATA-Zeilen in den Speicher geschrieben werden.

Doch bevor wir diesen Befehl ausführen, müssen wir kurz über den Befehl X zu BASIC wechseln, um den Arbeitsspeicher mit NEW zu löschen. Danach kehren wir durch Eingabe von SYS 49152 wieder zum SMON zurück.

```
-----  
.L"TEST"  
SEARCHING FOR TEST  
LOADING  
.D 3000  
.3000 A9 C8      LDA #C8  
.3002 A2 FF      LDX #FF  
.3004 A8 AC      LDY #AC  
.3006 69 37      ADC #37  
.3008 E8          INX  
.3009 C8          INY  
.300A 00          BRK  
-----  
.300B FF          ***  
.X  
READY.  
NEW  
  
READY.  
SYS 49152  
  
PC  SR  AC  XR  YR  SP  MU-BDIZC  
;C00B B0 C2 00 00 F6  10110000  
.■
```

Dies ist nötig, da ansonsten wieder der eingangs erwähnte Fehler OUT OF MEMORY angezeigt wird, wenn wir im Anschluss die BASIC-Zeilen für die benötigte FOR-Schleife eingeben wollen.

Doch nun ist alles vorbereitet und wir können uns von SMON die DATA-Zeilen durch Eingabe des Befehls B 3000 300B erzeugen lassen.

SMON generiert uns eine DATA Zeile, welche unser Programm als Zahlenfolge beinhaltet und springt ins BASIC zurück.

```
PC SR AC XR YR SP MU-BDIZC
;C00B B0 C2 00 00 F6 10110000
.B 3000 300B
32000D↑169,200,162,255,160,172,105,55,23
2,200,0
READY.
```

Sie wundern sich eventuell über die beiden Zeichen nach der Zeilennummer 32000.
Das D gefolgt von dem Pik-Zeichen ist die Abkürzung für die BASIC-Anweisung DATA.

Denn wenn wir nun LIST eingeben, wird uns folgende Zeile angezeigt:

```
PC SR AC XR YR SP MU-BDIZC
;C00B B0 C2 00 00 F6 10110000
.B 3000 300B
32000D↑169,200,162,255,160,172,105,55,23
2,200,0
READY.
LIST
32000 DATA169,200,162,255,160,172,105,55
232,200,0
READY.
```

Als nächstes ergänzen wir die FOR-Schleife, welche uns das Maschinenprogramm dann Byte für Byte in den Speicher schreibt.

```
2,200,0
READY.
LIST
32000 DATA169,200,162,255,160,172,105,55
232,200,0
READY.
10 FOR I=12288 TO 12298
20 READ A
30 POKE I,A
40 NEXT I
50 END
```

Das gesamte BASIC-Programm sieht dann so aus:

```
50 END
LIST

10 FOR I=12288 TO 12298
20 READ A
30 POKE I,A
40 NEXT I
50 END
32000 DATA 169,200,162,255,160,172,105,55
232,200,0
READY.
```

Wenn wir wollten, hätten wir hier sogar die Möglichkeit, die Adresse zu ändern ab der unser Programm im Speicher stehen soll. Dazu bräuchten wir nur die Startadresse 12288 bzw. die Endadresse 12298 in Zeile 10 entsprechend zu ändern.

Dieses BASIC-Programm können wir nun beispielsweise unter dem Namen „DATALOADER“ auf Diskette speichern.

Geben Sie also ein:

SAVE „DATALOADER“,8

```
SAVE "DATALOADER",8
SAVING DATALOADER
READY.
```

Nun starten wir unseren C64 neu und laden dieses BASIC-Programm mit dem Befehl

LOAD „DATALOADER“,8

wieder in den Speicher.

```
***** COMMODORE 64 BASIC V2 *****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
LOAD "DATALOADER",8
SEARCHING FOR DATALOADER
LOADING
READY.
LIST

10 FOR I=12288 TO 12298
20 READ A
30 POKE I,A
40 NEXT I
50 END
32000 DATA 169,200,162,255,160,172,105,55
232,200,0
READY.
```

Dann starten wir es mit RUN und das Maschinenprogramm wird nun wieder ab der Adresse 12288 (\$3000) in den Arbeitsspeicher geschrieben.

Laden Sie nun den SMON wieder in den Arbeitsspeicher, denn wir wollen nun erneut über das Disassemblieren prüfen, ob das Programm wieder im Speicher steht.

```
RUN  
READY.  
LOAD "SMONPC000",8,1  
SEARCHING FOR SMONPC000  
LOADING  
READY.  
NEW  
READY.  
SYS 49152  
PC SR AC XR YR SP NU-BDIZC  
;C00B B0 C2 00 00 F6 10110000  
.■
```

Und das Disassemblieren mit dem Befehl D 3000 beweist es – unser Programm steht nun wieder im Speicher!

```
PC SR AC XR YR SP NU-BDIZC  
;C00B B0 C2 00 00 F6 10110000  
.D 3000  
,3000 A9 C8 LDA #C8  
,3002 A2 FF LDX #FF  
,3004 A0 AC LDY #AC  
,3006 69 37 ADC #37  
,3008 E8 IMX  
,3009 C8 INY  
,300A 00 BRK  
-----  
.■
```

Nun wollen wir mal auf die andere Seite gehen und unser Maschinenprogramm von BASIC heraus mit SYS aufrufen.

Dazu ändern wir zuerst den Befehl BRK in den Befehl RTS, damit wir nach Beendigung des Programms im BASIC bleiben.

Anschließend wechseln wir mit dem Befehl „X“ ins BASIC zurück.

The screenshot shows a Commodore 64 assembly language debugger interface. At the top, there's a header with fields: PC, SR, AC, XR, YR, SP, and NU-BDIZC. Below this is a memory dump table with columns for Address (e.g., C00B, 3000, etc.), Value (e.g., B8, A9, FF, etc.), and Assembly (e.g., LDA #C8, LDX #FF, etc.). The assembly code listed is:

```

PC   SR  AC  XR  YR  SP  NU-BDIZC
;C00B B8  C2  00  00  F6  10110000
.D 3000
,3000  A9  C8      LDA #C8
,3002  A2  FF      LDX #FF
,3004  A0  AC      LDY #AC
,3006  69  37      ADC #37
,3008  E8          INX
,3009  C8         INY
,300A  60          RTS
-----
.X
READY .

```

Nun starten wir unser Programm mit SYS 12288 und sobald das Programm durchgelaufen ist, wollen wir die Inhalte des Akkumulators, des X Registers und des Y Registers prüfen.

Wie das aus BASIC heraus funktioniert, haben wir bereits gelernt. Durch das Auslesen der Speicherstellen 780, 781 und 782 erhalten wir die aktuellen Inhalte dieser drei Register.

Doch zunächst gehen wir das Programm durch und rechnen uns aus, welche Inhalte die Register aufweisen müssten.

Der erste Befehl lädt den Akkumulator mit dem dezimalen Wert 200, der zweite das X Register mit dem dezimalen Wert 255 und der dritte das Y Register mit dem dezimalen Wert 172.

Im nächsten Befehl wird der dezimale Wert 55 zum Inhalt des Akkumulators addiert, d.h. wir erhalten $200 + 55 = 255$.

Weiter geht's mit dem nächsten Befehl, der den Inhalt des X Registers um 1 erhöht, d.h. wir erhalten $255 + 1 = 256$.

Mathematisch soweit korrekt, aber das X Register kann ja nur Werte zwischen 0 und 255 fassen. Der Wert 256 ist jedoch bereits ein 16 Bit Wert, benötigt zur Darstellung also 2 Bytes.

Das X Register „läuft“ also über und springt wieder auf den Wert 0 zurück. Dasselbe würde passieren wenn es den Wert 0 enthielte und man 1 abziehen würde. Dann würde der Inhalt in die andere Richtung springen und 255 enthalten.

Mehr dazu im Kapitel über Zahlensysteme und Arithmetik.

Der nächste Befehl erhöht den Inhalt des Y Registers um 1, d.h. wir erhalten $172 + 1 = 173$.

```
:X  
READY.  
SYS 12288  
  
READY.  
PRINT PEEK (780)  
255  
  
READY.  
PRINT PEEK (781)  
0  
  
READY.  
PRINT PEEK (782)  
173  
  
READY.
```

Wie auf dem Screenshot zu sehen, decken sich die Werte an den Speicherstellen 780, 781 und 782 mit den Werten, die wir soeben für diese drei Register manuell ausgerechnet haben.

Ich hoffe, ich konnte Ihnen einen guten Einstieg in die Arbeit mit SMON verschaffen und Sie haben noch immer ungebrochenes Interesse daran, die Assembler-Sprache zu erlernen.

Im nächsten Kapitel starten wir nämlich mit der Programmierung durch.

Wir werden zunächst ein paar nützliche neue Befehle kennenlernen, uns näher mit dem Statusregister beschäftigen, Schleifen programmieren und uns auch mit den verschiedenen Adressierungsarten auseinandersetzen.

Das Gelernte werden wir anhand von zwei Übungen vertiefen, wobei die zweite Übung noch besser als die erste Übung, die Anwendung der Adressierungsarten die wir kennenlernen werden, zeigt.

Das Programm, welches Gegenstand der zweiten Übung ist, macht etwas sehr interessantes.

Nachdem es seine Aufgabe erledigt hat, teleportiert es sich selbst an eine andere Stelle im Speicher und löst sich selbst an seinem ursprünglichen Aufenthaltsort in Luft auf.

3 Von Adressierungsarten, Schleifen und Programmen, die sich selbst teleportieren

Zu Beginn möchte ich Ihnen einige neue Assembler-Befehle vorstellen, die wir später immer wieder mal brauchen werden und auch gut zu der Erklärung der Adressierungsarten passen, die wir gleich im Anschluss besprechen werden.

TAX

Dieser Befehl kopiert den Inhalt des Akkumulators in das X Register.

TXA

Umgekehrt kopiert dieser Befehl den Inhalt des X Registers in den Akkumulator.

TAY

Dieser Befehl kopiert den Inhalt des Akkumulators in das Y Register.

TYA

Umgekehrt kopiert dieser Befehl den Inhalt des Y Registers in den Akkumulator.

INC Speicheradresse

z.B. INC \$0400 erhöht den Inhalt der Speicherstelle \$0400 um 1.

DEC Speicheradresse

z.B. DEC \$0400 vermindert den Inhalt der Speicherstelle \$0400 um 1.

3.1 Adressierungsarten

Unter Adressierungsart versteht man die Art und Weise, wodurch die Quelle beschrieben wird, von der ein Wert gelesen wird bzw. wodurch das Ziel beschrieben wird, an das ein Wert geschrieben wird.

Die CPU des C64 beherrscht mehrere Adressierungsarten, von denen Sie einige, ohne es zu wissen, bereits kennengelernt haben.

3.1.1 Unmittelbare Adressierung

Bei dieser Art von Adressierung wird als Parameter eines Maschinenbefehls direkt ein Zahlenwert angegeben, welcher durch das vorangestellte # - Zeichen als solcher gekennzeichnet wird.

Hier ein Beispiel anhand des Befehls LDA, das uns schon bekannt ist:

LDA #\$C8

Dieser Befehl lädt den Akkumulator direkt mit dem Zahlenwert \$C8 (dezimal 200)

3.1.2 Absolute Adressierung

Von absoluter Adressierung spricht man, wenn als Parameter eines Maschinenbefehls eine 16 Bit Speicheradresse angegeben wird.

Hier ein ebenfalls bereits bekanntes Beispiel anhand des Befehls STX:

STX \$0400

Dieser Befehl schreibt den Inhalt des X Registers in die Speicherstelle mit der Adresse \$0400.

3.1.3 Zeropage Adressierung

Diese Adressierung gleicht der absoluten Adressierung. Die Besonderheit ist jedoch, dass die Adresse zwischen 0 und 255 liegt, also durch 1 Byte allein dargestellt werden kann.

Hier ein Beispiel anhand des Befehls LDY:

LDY \$64

Dieser Befehl lädt den Inhalt der Speicherstelle mit der Adresse \$64 (dezimal 100) in das Y Register.

Da ich soeben von der Zeropage Adressierung gesprochen habe, muss ich erklären, was sich hinter dem Begriff „Page“ verbirgt.

Unter einer Page versteht man einen Speicherblock welcher 256 Bytes umfasst.

Der Speicher des C64 besteht aus 65536 Speicherstellen, also aus $65356 / 256 = 256$ Pages, welche von 0 bis 255 durchnummeriert sind.

Die Page mit der Nummer 0 erstreckt sich von Adresse 0 bis Adresse 255 und wird daher Zeropage genannt.

3.1.4 Implizite Adressierung

Wenn im Maschinenbefehl selbst bereits alle Informationen zu dessen Ausführung enthalten sind, spricht man von impliziter Adressierung.

Gute Beispiele dafür haben wir gerade vorhin kennengelernt, z.B. die Befehle TAX, TXA, TAY und TYA.

Diese Befehle brauchen keine Parameter, weil in den Befehlen selbst bereits alle wichtigen Informationen enthalten sind. Der Befehl TAX enthält beispielsweise implizit die Quelle und das Ziel des Kopiervorganges (Quelle = Akkumulator, Ziel = X Register).

Weitere Beispiele wären die Befehle INX, INY, DEX und DEY, welche den Inhalt des jeweiligen Registers um eins erhöhen bzw. vermindern. Im Befehl selbst ist bereits enthalten, was zu tun ist. Im Falle des Befehls INX bedeutet dies, dass der Inhalt des X Registers um 1 erhöht werden soll.

3.1.5 X-indizierte absolute Adressierung

Diese Art der Adressierung funktioniert grundsätzlich gleich wie die absolute Adressierung. Der Unterschied ist, dass zur Adresse, welche dem Maschinenbefehl folgt, noch der Inhalt des X Registers addiert wird.

Hier ein Beispiel:

STA \$0400,X

Hier wird der Inhalt des Akkumulators an die Speicherstelle mit der Adresse \$0400 (dezimal 1024) + Inhalt des X Registers geschrieben. Angenommen, das X Register enthielte den Wert \$0A (dezimal 10), dann würde der Inhalt des Akkumulators an die Speicherstelle mit der Adresse \$040A (dezimal 1034) geschrieben werden.

3.1.6 Y-indizierte absolute Adressierung

Diese Art der Adressierung funktioniert grundsätzlich gleich wie die absolute Adressierung. Der Unterschied ist, dass zur Adresse, welche dem Maschinenbefehl folgt, noch der Inhalt des Y Registers addiert wird.

Hier ein Beispiel:

STA \$0400,Y

Hier wird der Inhalt des Akkumulators an die Speicherstelle mit der Adresse \$0400 (dezimal 1024) + Inhalt des Y Registers geschrieben. Angenommen, das Y Register enthielte den Wert \$0A (dezimal 10), dann würde der Inhalt des Akkumulators an die Speicherstelle mit der Adresse \$040A (dezimal 1034) geschrieben werden.

3.1.7 X-indizierte Zeropage Adressierung und Y-indizierte Zeropage Adressierung

Diese Adressierungsarten funktionieren analog zur X-indizierten bzw. Y-indizierten absoluten Adressierung, nur dass die Speicheradresse zwischen 0 und 255 liegt, also mit nur einem Byte dargestellt werden kann.

Es gibt noch einige weitere Adressierungsarten, doch die soeben beschriebenen Arten sollten für's Erste mal reichen.

3.2 Beispiel-Programme

Wir wollen nun unsere erste Schleife programmieren und bei dieser Gelegenheit werden wir auch gleich nach und nach Bekanntschaft mit dem Statusregister machen.

Sehen Sie sich folgendes Programm an:

```

.A 1500
1500 A2 00      LDX #00
1502 E8          INX
1503 E0 05      CPX #05
1505 D0 FB      BNE 1502
1507 00          BRK
1508 F
,1500 A2 00      LDX #00
,1502 E8          INX
,1503 E0 05      CPX #05
,1505 D0 FB      BNE 1502
,1507 00          BRK
-----
```

Es enthält zwei weitere neue Befehle: CPX und BNE.

Doch alles schön der Reihe nach und eine Kleinigkeit noch vorweg zur Anzeige der Zahlen im SMON.

Wir geben die Zahlenwerte zwar immer mit dem vorangestellten \$ - Zeichen ein, aber der SMON zeigt dieses \$ - Zeichen dann nicht an, sondern einfach nur die hexadezimale Zahl ohne das vorangestellte \$ - Zeichen.

Hier wird zuerst das X Register mit dem Wert 0 geladen. Als nächstes wird dessen Inhalt um 1 erhöht, d.h. es enthält nun den Wert 1.

Nun kommt der erste neue Befehl:

CPX #05 (einzugeben als CMP #\$05)

CPX steht für Compare X.

Durch ihn weisen wir die CPU an, den aktuellen Inhalt des X Registers mit dem Wert 5 zu vergleichen. Die CPU führt den Vergleich durch, indem sie intern den Wert 5 vom aktuellen Inhalt des X Registers abzieht. Solange der Inhalt des X Registers ungleich 5 ist, wird das Ergebnis dieser Subtraktion immer einen Wert ungleich 0 ergeben.

Falls das X Register jedoch den Wert 5 enthält, dann ergibt die Subtraktion von 5 den Wert 0.

Nun kommt das Statusregister ins Spiel.

Die einzelnen Bits des Statusregisters nennt man auch Flags und eines dieser Flags ist das sogenannte Zeroflag (Bit 1). Es wird von der CPU immer dann auf 1 gesetzt, wenn das Ergebnis einer Aktion gleich 0 war.

Genau dies trifft auf den oben erwähnten Fall zu: $5 - 5 = 0$, d.h. die CPU setzt das Zeroflag auf 1. In den anderen Fällen, in denen das Ergebnis ungleich 0 ist, setzt die CPU das Zeroflag auf 0.

Nun kommt der nächste neue Befehl:

BNE xxxx

BNE steht für **Branch on Not Equal** (verzweige bei Ungleichheit zur Adresse xxxx, in diesem Fall zur Adresse \$1502)

Das Programm wird im Falle eines nicht gesetzten ZeroFlags also an der Adresse \$1502 fortgesetzt.

Was steht an der Adresse \$1502 im Speicher? Dort steht der Befehl INX, welcher den Inhalt des X Registers um 1 erhöht, d.h. es enthält nun den Wert 2.

Als nächstes wird wiederum der Vergleich des Inhalts des X Registers mit 5 durchgeführt. Der aktuelle Inhalt 2 ist ungleich 5, d.h. wir verzweigen wieder zur Adresse \$1502.

Dort wird der Inhalt des X Registers wieder um 1 erhöht, d.h. es enthält nun den Wert 3.

Das geht solange weiter, bis das X Register den Wert 5 enthält.

In diesem Fall ist das Ergebnis der Subtraktion, welche die CPU intern durchführt, gleich 0, d.h. sie setzt das ZeroFlag auf 1.

Nun findet kein Sprung mehr an die Adresse \$1502 statt, weil die Bedingung dafür ja nicht mehr erfüllt ist.

Stattdessen wird die Ausführung des Programms direkt mit dem Befehl, welcher auf den Befehl BNE folgt fortgesetzt. In diesem Fall ist das der Befehl BRK, durch den das Programm beendet wird.

Starten Sie das Programm doch mal mit dem Befehl G 1500 und wir erhalten folgende Anzeige:

The screenshot shows a Z80 assembly debugger interface. At the top, it says ".G 1500". Below that, there's a register dump table with columns: PC, SR, AC, XR, YR, SP, and NU-BDIZC. The values are: PC: 1509, SR: 33, AC: C2, XR: 05, YR: 00, SP: F6, NU-BDIZC: 00110011. There are two small blue squares below the table.

Hier sieht man, dass das X Register den Wert 5 aufweist und Bit 1 (Z wie ZeroFlag) gesetzt ist.

Möglicherweise ist Ihnen bei der Übersetzung des Befehls BNE 1502 in Maschinencode etwas seltsam erschienen.

The screenshot shows assembly code in a debugger. It includes labels .1503, .1505, .1507, E0, 05, D0, FB, CPX #05, BNE 1502, and RRK. The labels .1503, .1505, and .1507 are aligned with the first three bytes of the instruction, while E0, 05, D0, FB are aligned with the next four bytes, and CPX #05, BNE 1502, and RRK are aligned with the final byte.

Hier sieht man, dass der Befehl in die hexadezimale Zahlenfolge \$D0 \$FB übersetzt wurde.

Der Code \$D0 lässt sich erklären, denn dies ist der Befehlscode für den Befehl BNE, aber was hat es mit dem Wert \$FB auf sich? Es ist hier kein Zusammenhang zum Wert \$1502 erkennbar. Die Erklärung ist folgende:

Der Maschinenbefehl BNE erwartet als Parameter eigentlich keine Adresse, sondern eine Entfernungsangabe relativ zum aktuellen Maschinenbefehl.

Dass wir hier nach dem Befehl BNE eine Adresse eingeben können, ist ein hilfreiches Feature des Assemblers, der bei der Übersetzung die Differenz zwischen der aktuellen Adresse und jener Adresse, die angesprungen werden soll, berechnet und entsprechend den korrekten Maschinencode erzeugt.

Den Wert \$FB kann man als den dezimalen Wert 251 interpretieren, aber auch als den dezimalen Wert -5 (siehe Kapitel über die Zahlensysteme)

Der Wert -5 sagt der CPU, wieviele Bytes sie zum Inhalt des Program Counter Registers addieren muß.

In diesem Fall $\$1507 - 5 = \1502

Bei einem Sprung nach vorne wäre die Entfernungsangabe entsprechend positiv.

Hinweis:

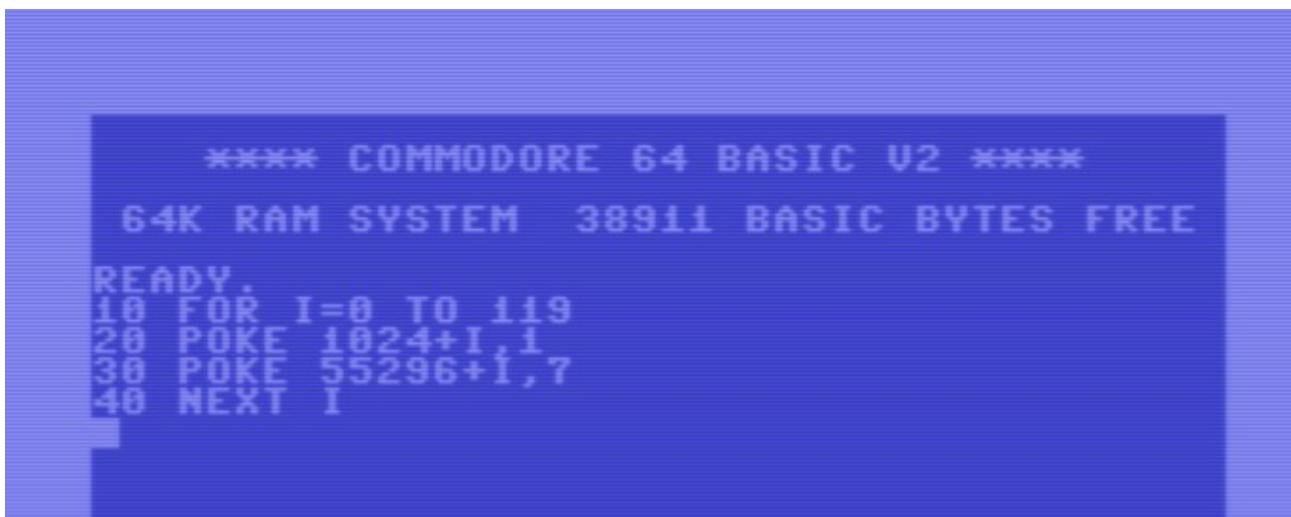
Es gibt auch Compare-Befehle für den Akkumulator und das Y Register, diese heißen CMP für den Akkumulator und CPY für das Y Register.

Auch für den Befehl BNE gibt es ein Gegenstück, er heißt BEQ (Branch On Equal). Die Verzweigung findet also nicht aufgrund von Ungleichheit, sondern aufgrund von Gleichheit statt.

In den folgenden Kapiteln werden wir noch weitere Branch Befehle kennenlernen.

Nun werde ich Ihnen demonstrieren, wie schnell ein Assembler-Programm im Vergleich zu einem BASIC-Programm ist.

Geben Sie also zunächst mal folgendes BASIC-Programm ein und speichern es am besten mit SAVE „BASICDEMO“,8 auf Diskette.



The screenshot shows the Commodore 64 BASIC V2 interface. The title screen displays "***** COMMODORE 64 BASIC V2 *****" and "64K RAM SYSTEM 38911 BASIC BYTES FREE". Below this, the text "READY." is shown. A series of BASIC commands are listed, starting with line 10, which contains a FOR loop from I=0 to 119. The commands are as follows:

```
10 FOR I=0 TO 119
20 POKE 1024+I,1
30 POKE 55296+I,7
40 NEXT I
```

Was macht das Programm?

Das ist schnell beschrieben:

Es beschreibt die ersten drei Zeilen des Bildschirms mit einem gelben A.

Wie wird das gemacht?

Wir wollen drei Zeilen vollschreiben, das entspricht 120 Zeichen, d.h. wir erstellen eine FOR-Schleife mit 120 Durchläufen und schreiben das A in die entsprechenden Speicherstellen des Videospeichers, den wir bereits kennengelernt haben.

Die Adresse \$0400 (dezimal 1024) entspricht der linken oberen Ecke des Bildschirms.

Im ersten Schleifendurchlauf ist I gleich 0, d.h. wir schreiben den Zeichencode des Buchstabens A in die Speicherstelle 1024 + 0, also in die Speicherstelle 1024.

Im zweiten Schleifendurchlauf ist I gleich 1, d.h. wir schreiben den Zeichencode des Buchstabens A in die Speicherstelle 1024 + 1, also in die Speicherstelle 1025.

Das setzt sich fort bis zum letzten Durchlauf, in dem I gleich 119 ist. In diesem letzten Durchlauf wird der Zeichencode des Buchstabens A in die Speicherstelle 1024 + 119 = 1143 (\$0477) geschrieben, welche der letzten Position in der dritten Zeile entspricht.

Doch was hat es mit dieser Speicheradresse 55296 (\$D800) auf sich? Dies ist der Beginn des Farbspeichers des C64, welcher die Farbcodes der Zeichen enthält, welche am Bildschirm angezeigt werden, d.h. auch der Farbspeicher umfasst 1000 Bytes, in denen für jedes der 1000 Zeichen am Bildschirm einer der folgenden Farbcodes gespeichert ist:

| Farbe | Name | Farbwert für POKE | HEX |
|-------|----------|-------------------|------|
| ■ | Schwarz | 0 | \$00 |
| ■ | Weiß | 1 | \$01 |
| ■ | Rot | 2 | \$02 |
| ■ | Türkis | 3 | \$03 |
| ■ | Violett | 4 | \$04 |
| ■ | Grün | 5 | \$05 |
| ■ | Blau | 6 | \$06 |
| ■ | Gelb | 7 | \$07 |
| ■ | Orange | 8 | \$08 |
| ■ | Braun | 9 | \$09 |
| ■ | Hellrot | 10 | \$0a |
| ■ | Grau 1 | 11 | \$0b |
| ■ | Grau 2 | 12 | \$0c |
| ■ | Hellgrün | 13 | \$0d |
| ■ | Hellblau | 14 | \$0e |
| ■ | Grau 3 | 15 | \$0f |

Der Wert in der Speicherstelle 55296 enthält den Farbcode jenes Zeichens, dessen Zeichencode in der Speicherstelle 1024 gespeichert ist, also die Farbe des Zeichens in der linken oberen Bildschirmecke.

In der Speicherstelle 55297 steht der Farbcode des Zeichens rechts daneben und so weiter bis zur rechten unteren Bildschirmecke.

Da wir den Buchstaben A immer in gelber Farbe haben wollen, schreiben wir jeweils immer den Wert 7 an die entsprechenden Stelle im Farbspeicher.

Das letzte Zeichen in der dritten Zeile hat im Bildschirmspeicher die Adresse $1024 + 119 = 1143$ (\$0477) und der zugehörige Farbcode steht in der Speicherstelle $55296 + 119 = 55415$ (\$D877)

Löschen Sie den Bildschirm mit SHIFT + CLR/HOME, bewegen den Cursor um einige Zeilen nach unten und starten das Programm mit RUN.

Das Programm füllt die ersten drei Zeilen des Bildschirms mit dem Buchstaben A in gelber Farbe.



Ich habe mal mitgestoppt, es dauert ca. 3 Sekunden, bis das Programm durchgelaufen ist.

Nun werden wir das gleiche Programm in Assembler schreiben. Starten Sie also den SMON und geben folgendes Assembler-Programm ab der Adresse \$1500 (dezimal 5376) ein.

```

PC  SR  AC  XR  YR  SP  MU-BDIZC
;C00B  B0  C2  00  00  F6  10110000
.A 1500
1500  A2  00      LDX #00
1502  A9  01      LDA #01
1504  9D  00  04  STA 0400,X
1507  A9  07      LDA #07
1509  9D  00  D8  STA D800,X
150C  E8          INX
150D  E0  78      CPX #78
150F  D0  F1      BNE 1502
1511  00          BRK
1512  F
,1500  A2  00      LDX #00
,1502  A9  01      LDA #01
,1504  9D  00  04  STA 0400,X
,1507  A9  07      LDA #07
,1509  9D  00  D8  STA D800,X
,150C  E8          INX
,150D  E0  78      CPX #78
,150F  D0  F1      BNE 1502
,1511  00          BRK
-----.
■

```

Speichern Sie das Programm mit dem Befehl

```

S"ASSEMBLERDEMO" 1500 1512
SAVING ASSEMBLERDEMO
■

```

auf Diskette ab.

Im ersten Befehl wird das X Register mit dem Wert 0 geladen und mit dem nächsten Befehl der Akkumulator mit dem Wert 1, denn dies ist der Zeichencode für den Buchstaben A. Diese zwei Befehle sind gute Beispiele für die unmittelbare Adressierung.

Durch den Befehl STA 0400,X wird dieser Wert in die Speicherstelle \$0400 + Inhalt des X Registers geschrieben, also in die Speicherstelle \$0400 + 0 = \$0400 (1024 dezimal), d.h. in die linke obere Ecke des Bildschirms.

Nun wird der Akkumulator mit dem Wert 7 geladen, denn dies ist der Farocode für Gelb. Auch dieser Befehl ist ein Beispiel für die unmittelbare Adressierung (LDA #07).

Durch den Befehl STA D800,X wird dieser Wert in die Speicherstelle \$D800 + Inhalt des X Registers geschrieben, also in die Speicherstelle \$D800 + 0 = \$D800 (55296 dezimal)

Somit ist schon mal das erste gelbe A in der linken oberen Ecke des Bildschirms zu sehen.

Die Befehle STA 0400,X und STA D800,X stellen Beispiele für die X indizierte absolute Adressierung dar.

Als nächstes wird der Inhalt des X Registers um 1 erhöht und durch den Befehl CPX wird dessen Inhalt mit dem Wert \$78 (120 dezimal) verglichen. Auch dieser Befehl ist ein Beispiel für die unmittelbare Adressierung.

Solange der Wert des X Registers nicht den Wert 120 erreicht hat, wird durch die Anweisung BNE 1502 wieder zur Programmadresse \$1502 gesprungen.

Dort wird der Akkumulator wieder mit dem Zeichencode für den Buchstaben A geladen und dieser dann an die Speicherstelle \$0400 + 1 (Inhalt des X Registers) = \$0401 (dezimal 1025) geschrieben.

Dann wird der Akkumulator wieder mit dem Farbcode für Gelb geladen und dieser in die Speicherstelle \$D800 + 1 (Inhalt des X Registers) = \$D801 (dezimal 55297) in den Farbspeicher geschrieben.

Der nächste Befehl INX erhöht den Inhalt des X Registers wieder um 1, d.h. es hat nun den Inhalt 2. Durch den Befehl INX haben wir hier auch ein Beispiel für die implizite Adressierung.

Der folgende Vergleich mit dem Wert 120 bringt wieder Ungleichheit, d.h. es wird wieder zur Programmadresse \$1502 gesprungen.

Auf diese Art und Weise wird ein A nach dem anderen in den Bildschirmspeicher geschrieben, bis die drei Zeilen vollgeschrieben sind.

Zu jedem A wird auch der Farbcode für Gelb in die entsprechende Speicherstelle im Farbspeicher geschrieben, sodass jedes ausgegebene A in gelber Farbe erscheint.

Das X Register hat in unserem Assembler-Programm hier dieselbe Aufgabe wie die Schleifen-Variable I in unserem BASIC-Programm. Der Inhalt des X Registers wird ebenfalls von 0 bis 119 hochgezählt.

Nachdem das letzte A ausgegeben wurde, enthält das X Registers den Wert 119. Durch den Befehl INX erhält es den Wert 120 und nun bringt der Vergleich durch den Befehl CPX Gleichheit, d.h. es wird nun nicht mehr zur Programmadresse \$1502 verzweigt, sondern mit dem direkt folgenden Befehl fortgesetzt.

In diesem Fall ist das der Befehl BRK, d.h. das Programm wird beendet.

Löschen Sie nun wiederum den Bildschirm mit SHIFT + CLR/HOME, bewegen den Cursor einige Zeilen nach unten und geben ein:

.G 1500 (vergessen Sie den Punkt nicht, denn er ist durch das Löschen des Bildschirms ebenfalls verschwunden)



Haben Sie versucht die Zeit mitzustoppen? Vergessen Sie es, das Programm ist durchgelaufen noch ehe Sie überhaupt die Stoppuhr starten können.

Dieses Assembler-Programm und das vorherige BASIC-Proggramm tun exakt dasselbe, aber dieses Beispiel hat eindrucksvoll bewiesen, wie schnell das Assemblerprogramm im Vergleich zu der BASIC-Variante ist.

Sie können gerne zum Vergleich nochmal das BASIC-Programm BASICDEMO von der Diskette laden und mit RUN starten.

Das Assembler-Programm ist nicht nur schneller, es beansprucht nur ganze 18 Bytes im Speicher. Wieviel Speicher das BASIC-Programm genau verbraucht, müsste man nachrechnen, aber wenn man nachprüft, um wieviele Bytes sich der freie Arbeitsspeicher nach der Eingabe des BASIC-Programms reduziert hat, dann sind es mindestens 57 Bytes, also mehr als dreimal soviel wie das Assembler-Programm!



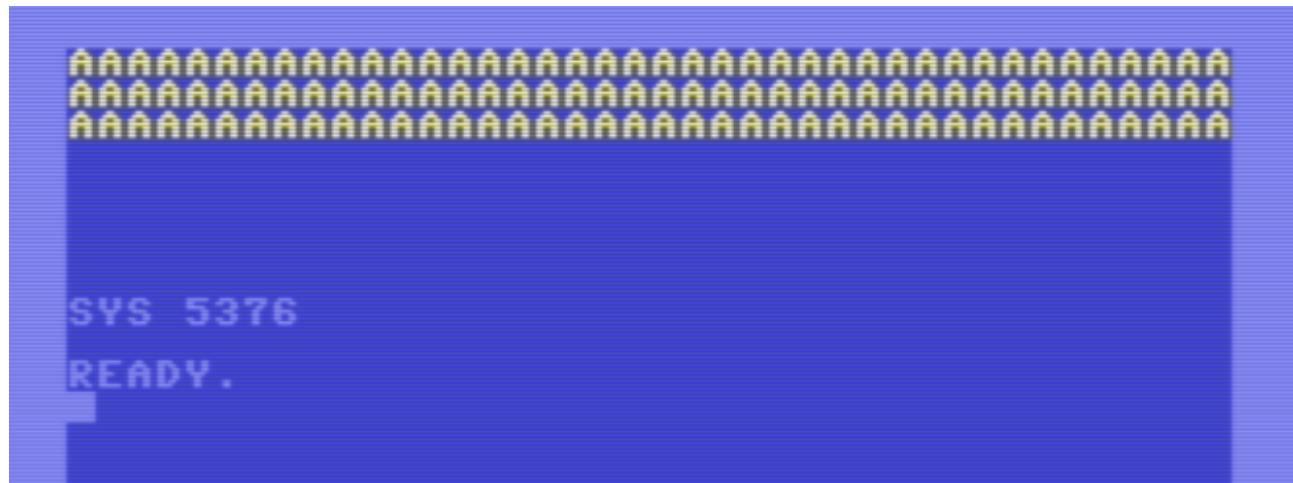
Starten wir das Maschinenprogramm zur Abwechslung mal wieder von BASIC aus. Dazu müssen Sie den BRK Befehl am Ende des Programms in den RTS Befehl ändern, damit wir nach dem Beenden des Programms im BASIC bleiben.

Danach kehren Sie mit dem SMON-Befehl „X“ ins BASIC zurück.

The screenshot shows the Commodore 64 monitor displaying assembly language code. The code is as follows:

```
.G 1500
    PC   SR  AC  XR  YR  SP  NU-BDIZC
; 1512 33 07 78 00 F6  00110011
.D 1500
, 1500 A2 00      LDX #00
, 1502 A9 01      LDA #01
, 1504 9D 00 04    STA 0400,X
, 1507 A9 07      LDA #07
, 1509 9D 00 D8    STA D800,X
, 150C E8          INX
, 150D E0 78      CPX #78
, 150F D0 F1      BNE 1502
, 1511 60          RTS
.
.X
READY.
```

Löschen Sie nun wieder den Bildschirm mit SHIFT + CLR/HOME, bewegen den Cursor um einige Zeilen nach unten und starten das Maschinenprogramm durch SYS 5376.



Das Ergebnis ist klarerweise dasselbe, nur dass Sie diesesmal das Programm aus BASIC heraus mit SYS 5376 gestartet haben und nicht aus SMON heraus mit .G 1500

Erstellen wir zur Übung einen BASIC-Loader für unser Programm? Aber sicher.

Um den besagten Fehler OUT OF MEMORY ERROR zu vermeiden, geben wir wieder den Befehl NEW ein und wechseln erst dann mit SYS 49152 wieder zu SMON zurück.

Dort erstellen durch Eingabe des Befehls

B 1500 1512

eine DATA-Zeile, welche unser Maschinenprogramm als Zahlenfolge enthält.

READY.
SYS 49152

| PC | SR | AC | XR | YR | SP | NU-BDIZC |
|--------|------|--|----|----|----|----------|
| :C00B | B1 | C2 | 78 | 00 | F2 | 10110001 |
| B 1500 | 1512 | | | | | |
| 32000D | DATA | 162,0,169,1,157,0,4,169,7,157,0,2 16,216,232,224,120,208,241,96 | | | | |
| | | | | | | |
| | | | | | | READY. |

Ergänzen wir nun die FOR-Schleife, damit wir unser Maschinenprogramm durch den BASIC-Loader in den Speicher ab 5376 schreiben können.

Speichern Sie das Programm mit SAVE „BASICLOADER“,8 auf Diskette ab.

LIST

```
32000 DATA162,0,169,1,157,0,4,169,7,157,  
0,216,232,224,120,208,241,96
```

READY.

```
10 FOR I=5376 TO 5393  
20 READ A  
30 POKE I,A  
40 NEXT I  
50 END
```

LIST

```
10 FOR I=5376 TO 5393  
20 READ A  
30 POKE I,A  
40 NEXT I  
50 END
```

```
32000 DATA162,0,169,1,157,0,4,169,7,157,  
0,216,232,224,120,208,241,96
```

READY.

```
SAVE "BASICLOADER",8
```

SAVING BASICLOADER

READY.

Starten Sie nun Ihren C64 neu und laden das Programm wieder mit LOAD „BASICLOADER“,8 und starten den Loader mit RUN.

```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
LOAD "BASICLOADER",8
SEARCHING FOR BASICLOADER
LOADING
READY.
LIST

10 FOR I=5376 TO 5393
20 READ A
30 POKE I,A
40 NEXT I
50 END
32000 DATA 162,0,169,1,157,0,4,169,7,157,
0,216,232,224,120,208,241,96
READY.
RUN

READY.
```

Unser Maschinenprogramm befindet sich nun wieder ab Adresse 5376 im Speicher.

Löschen wir wie gehabt wieder den Bildschirm mit SHIFT + CLR/HOME, bewegen den Cursor um einige Zeilen nach unten und führen den Befehl SYS 5376 aus.

Und wir sehen: Es funktioniert!



Laden Sie nun den SMON, löschen den BASIC-Loader mit NEW und starten den SMON mit SYS 49152.

Speichern Sie das Maschinenprogramm unter dem Namen „THREELINES“ auf Diskette, denn wir werden es nun für zwei Übungen benötigen, in denen Sie alles anwenden, was Sie bis jetzt gelernt haben.

The screenshot shows the Commodore 64 assembly language editor. The code is as follows:

```
PC SR AC XR YR SP NU-BDIZC
;C000B B0 C2 00 00 F6 10110000
.D 1500
,1500 A2 00 LDX #00
,1502 A9 01 LDA #01
,1504 9D 00 04 STA $400,X
,1507 A9 07 LDA #07
,1509 9D 00 D8 STA $800,X
,150C E8 INX
,150D E0 78 CPX #78
,150F D0 F1 BNE 1502
,1511 60 RTS
-----
.S "THREELINES" 1500 1512
SAVING THREELINES
■
```

Starten Sie den C64 neu und laden das Maschinenprogramm wieder in den Speicher.

The screenshot shows the Commodore 64 assembly language editor. The code is identical to the previous one, but the screen also displays the loading process:

```
PC SR AC XR YR SP NU-BDIZC
;C000B B0 C2 00 00 F2 10110000
.L "THREELINES"
SEARCHING FOR THREELINES
LOADING
.D 1500
,1500 A2 00 LDX #00
,1502 A9 01 LDA #01
,1504 9D 00 04 STA $400,X
,1507 A9 07 LDA #07
,1509 9D 00 D8 STA $800,X
,150C E8 INX
,150D E0 78 CPX #78
,150F D0 F1 BNE 1502
,1511 60 RTS
-----
■
```

Die erste Übung besteht darin, die obersten drei Zeilen des Bildschirms, welche das Programm ja mit dem Buchstaben A gefüllt hat, in die untersten drei Zeilen des Bildschirms zu kopieren. Allerdings sollen diese A's dann nicht in Gelb, sondern in Türkis (Farocode 3), angezeigt werden.

Dazu müssen wir das Programm erweitern und weitere Befehle nach dem Befehl BNE 1502 ergänzen. Die neuen Befehle beginnen im Speicher also an jener Adresse, an der aktuell der Befehl RTS steht, also an der Adresse \$1511.

Um Befehle ab der Adresse \$1511 in den Speicher zu schreiben, geben wir den Befehl A 1511 ein.

```

,150F D8 F1      BNE 1502
,1511 60          RTS
-----
.A 1511
1511 A2 00        LDX #00
1513 BD 00 04    LDA 0400,X
1516 9D 70 07    STA 0770,X
1519 A9 03        LDA #03
151B 9D 70 DB    STA DB70,X
151E E8          INX
151F E0 78        CPX #78
1521 D8 F0        BNE 1513
1523 60          RTS
1524 F
,1511 A2 00        LDX #00
,1513 BD 00 04    LDA 0400,X
,1516 9D 70 07    STA 0770,X
,1519 A9 03        LDA #03
,151B 9D 70 DB    STA DB70,X
,151E E8          INX
,151F E0 78        CPX #78
,1521 D8 F0        BNE 1513
,1523 60          RTS
-----
.■

```

Wir ergänzen die zusätzlichen Befehle ab Adresse \$1511 beginnend mit dem Befehl LDX #\$00 und endend mit dem Befehl RTS an der Adresse \$1523.

Wechseln Sie mit dem SMON-Befehl „X“ zurück nach BASIC, löschen den Bildschirm, bewegen den Cursor einige Zeilen nach unten und starten das Programm mit SYS 5376.

Wenn alles richtig gelaufen ist, sollte das Ergebnis folgendermaßen aussehen:



Wechseln Sie nun mit SYS 49152 wieder zurück zu SMON und speichern das Programm unter dem Namen „SIXLINES“ auf Diskette, denn es wird als Basis für die zweite Übung dienen.

```
.D 1500
,1500 A2 00      LDX #00
,1502 A9 01      LDA #01
,1504 9D 00 04    STA 0400,X
,1507 A9 07      LDA #07
,1509 9D 00 D8    STA D800,X
,150C E8          INX
,150D E0 78      CPX #78
,150F D0 F1      BNE 1502
,1511 A2 00      LDX #00
,1513 BD 00 04    LDA 0400,X
,1516 9D 70 07    STA 0770,X
,1519 A9 03      LDA #03
,151B 9D 70 DB    STA DB70,X
,151E E8          INX
,151F E0 78      CPX #78
,1521 D0 F0      BNE 1513
,1523 60          RTS
-----
.S"SIXLINES" 1500 1524
SAVING SIXLINES
.■
```

Und diese wird spannend, denn das Programm soll sich, nachdem es beendet wurde, an eine andere Stelle im Arbeitsspeicher teleportieren, d.h. sich selbst an eine andere Stelle im Speicher kopieren und sich danach an der ursprünglichen Stelle quasi in Luft auflösen.

Doch bevor wir damit beginnen, gehen wir zunächst das erste Übungsprogramm Byte für Byte durch.

Die Änderung beginnt ab Adresse \$1511, hier stand vorher der Befehl RTS, doch da wir unser Programm ja erweitert haben, musste er dem ersten neu hinzugefügten Befehl weichen.

Dieser initialisiert das X Register mit dem Wert 0, welches auch hier wieder als Schleifenzähler fungiert.

Wir wollen die Inhalte der obersten 3 Zeilen in die unteren 3 Zeilen kopieren.

Dazu durchlaufen wir alle Zeichen vom Beginn der ersten Zeile bis zum letzten Zeichen der dritten Zeile.

Umgelegt auf Speicheradressen lesen wir nacheinander die Adressen \$0400 (dezimal 1024) bis \$0477 (dezimal 1143) im Videospeicher aus und im Akkumulator landet dabei immer der Wert 1, da dies ja der Zeichencode für den Buchstaben A ist.

Wir verwenden dazu den Befehl

LDA \$0400,X

Das bedeutet, wir verwenden als Basisadresse immer die Adresse \$0400, wobei jedoch immer der aktuelle Inhalt des X Registers hinzugezählt wird, um auf die wirkliche Adresse zu kommen, aus der gelesen wird.

Der nächste Befehl

STA \$0770,X

funktioniert ähnlich, nur dass dieses mal nicht aus einer Speicherstelle gelesen wird, sondern in eine Speicherstelle geschrieben wird. In die Zieladresse wird ebenfalls immer der Wert 1 geschrieben, da im Akkumulator ja nach jedem Lesevorgang eine 1 steht.

Auch beim Schreiben verwenden wir hier die X indizierte absolute Adressierung, wobei die Basisadresse die Adresse \$0770 (dezimal 1904) ist, zu der bei jedem Schleifendurchlauf der aktuelle Inhalt des X Registers hinzugezählt wird, um auf die Adresse zu kommen, an die geschrieben wird.

Zu Beginn ist dies die Adresse im Videospeicher, welche das erste Zeichen der untersten 3 Zeilen darstellt.

Rechnen wir nach:

Die Zeilen 23, 24 und 25 stellen die untersten 3 Zeilen dar. Darüber liegen 22 Zeilen mit jeweils 40 Zeichen. Das ergibt 880 Bytes und diese Anzahl müssen wir noch zur Startadresse des

Videospeichers hinzuzählen, womit wir auf $1024 + 880 = 1904$ oder eben \$0770 in hexadezimaler Schreibweise kommen.

Bei jedem Schleifendurchlauf wird der Inhalt des Akkumulators in die Speicherstelle \$0770 + Inhalt des X Registers geschrieben.

Nun müssen wir noch für jedes Zeichen der untersten 3 Zeilen die Farbinformation in den Farbspeicher schreiben.

Dazu müssen wir zunächst den Farbcode 3 für Türkis in den Akkumulator laden:

LDA #\$03

Diesen Wert speichern wir nach demselben Schema über die X indizierte absolute Adressierung in den Farbspeicher.

STA \$DB70,X

Die Berechnung der Bezugsadresse funktioniert gleich wie vorhin.

Die Startadresse des Farbspeichers ist \$D800 (dezimal 55296). Zu dieser Startadresse müssen wir wieder 880 Bytes hinzuzählen, womit wir auf die Adresse \$DB70 (dezimal 56176) kommen.

In dieser Speicherstelle steht nun der Farbcode, welcher zum ersten Zeichen der untersten 3 Zeilen gehört.

Hier zum besseren Verständnis eine Darstellung, was hier während dem Durchlaufen der Schleife geschieht:

| Inhalt des X Registers | Speicherstelle im Videospeicher aus der gelesen wird | Speicherstelle im Videospeicher in die der ausgelesene Zeichencode geschrieben wird | Speicherstelle im Farbspeicher in die der Farbcode für das kopierte Zeichen geschrieben wird |
|------------------------|--|---|--|
| 0 | \$0400 + 0 = \$0400 | \$0770 + 0 = \$0770 | \$DB70 + 0 = \$DB70 |
| 1 | \$0400 + 1 = \$0401 | \$0770 + 1 = \$0771 | \$DB70 + 1 = \$DB71 |
| 2 | \$0400 + 2 = \$0402 | \$0770 + 2 = \$0772 | \$DB70 + 2 = \$DB72 |
| . | | | |
| . | | | |
| . | | | |
| 119 | \$0400 + 119 = | \$0770 + 119 = | \$DB70 + 119 = \$DBE7 |

| Inhalt des X Registers | Speicherstelle im Videospeicher aus der gelesen wird | Speicherstelle im Videospeicher in die der ausgelesene Zeichencode geschrieben wird | Speicherstelle im Farbspeicher in die der Farbcode für das kopierte Zeichen geschrieben wird |
|------------------------|--|---|--|
| | \$0477 | \$07E7 | |

Nachdem die Schleife durchlaufen wurde, sind die untersten 3 Zeilen mit einem A in türkiser Farbe gefüllt.

Kommen wir nun zur zweiten Übung. Hier wird das soeben erweiterte Programm nochmals erweitert. Bei den bisherigen Programmen habe ich Ihnen Schritt für Schritt erklärt, wie das Programm funktioniert.

Diese detaillierte Erklärung lasse ich dieses mal bewusst weg, denn ich möchte Sie dazu ermutigen, selbst zu versuchen, hinter die Funktionsweise des Programms zu kommen :)

Nachdem die obersten und untersten 3 Zeilen mit den A's befüllt wurden, soll sich das Programm im ersten Schritt selbst an eine andere Speicheradresse kopieren, es findet also die Teleportation statt :)

Nehmen wir beispielsweise die Adresse \$3000 (dezimal 12288)

Nach dem Kopieren seines Programmcodes soll es im zweiten Schritt seinen Programmcode mit Nullwerten überschreiben, sodass nur mehr der Maschinencode übrig bleibt, der das Überschreiben durchführt.

Zunächst müssen wir abzählen, wie viele Bytes unser Programm im Speicher belegt, damit wir wissen, wie viele Bytes wir kopieren müssen.

Ich habe 36 Bytes gezählt, Byte Nr. 1 steht an der Adresse \$1500 (dezimal 5376) und Byte Nr. 36 steht an der Adresse \$1523 (dezimal 5411)

Hier nochmal zur Hilfe und zum Nachzählen der Screenshot von vorhin:

The screenshot shows a memory dump from address \$1500 to \$1523. The dump is as follows:

| Adresse | Wert | Befehl |
|---------|----------|------------|
| \$1500 | A2 00 | LDX #00 |
| \$1502 | A9 01 | LDA #01 |
| \$1504 | 9D 00 04 | STA 0400,X |
| \$1507 | A9 07 | LDA #07 |
| \$1509 | 9D 00 D8 | STA D800,X |
| \$150C | E8 | INX |
| \$150D | E0 78 | CPX #78 |
| \$150F | D8 F1 | BNE 1502 |
| \$1511 | A2 00 | LDX #00 |
| \$1513 | BD 00 04 | LDA 0400,X |
| \$1516 | 9D 70 07 | STA 0770,X |
| \$1519 | A9 03 | LDA #03 |
| \$151B | 9D 70 DB | STA DB70,X |
| \$151E | E8 | INX |
| \$151F | E0 78 | CPX #78 |
| \$1521 | D8 F0 | BNE 1513 |
| \$1523 | 60 | RTS |

Below the dump, there is a dashed line followed by the text:

.S"SIXLINES" 1500 1524
SAVING SIXLINES
.■

An Adresse \$1523 steht aktuell der Befehl RTS.

Da wir unser Programm erweitern wollen, müssen wir diesen, wie vorhin, mit dem ersten neu hinzugefügten Befehl überschreiben.

Durch den Befehl A 1523 werden die neuen Befehle ab der Adresse \$1523 gespeichert und daher der bisherige Befehl RTS überschrieben.

Geben Sie also die folgenden neuen Befehle ein und beenden die Eingabe wie gehabt mit dem Befehl „F“, sodass Sie folgende Anzeige erhalten:

```

,1521 D0 F0      BNE 1513
,1523 60          RTS
-----
. A 1523
1523 A2 00      LDX #00
1525 BD 00 15    LDA 1500,X
1528 9D 00 30    STA 3000,X
152B E8          INX
152C E0 23      CPX #23
152E D0 F5      BNE 1525
1530 A9 60      LDA #60
1532 8D 23 30    STA 3023
1535 60          RTS
1536 F
,1523 A2 00      LDX #00
,1525 BD 00 15    LDA 1500,X
,1528 9D 00 30    STA 3000,X
,152B E8          INX
,152C E0 23      CPX #23
,152E D0 F5      BNE 1525
,1530 A9 60      LDA #60
,1532 8D 23 30    STA 3023
,1535 60          RTS
-----
■

```

Erklärungsbedarf besteht hier glaube ich bei dem Befehl CPX #23 an der Adresse \$152C
\$23 entspricht dem dezimalen Wert 35.

Wenn das X Register den Wert 35 enthält, dann wurden erst 35 Bytes kopiert, nämlich jene von Adresse \$1500 bis Adresse \$1522. Warum kopieren wir also das Byte an der Adresse \$1523, also das letzte von den 36 Bytes nicht mit?

Der Grund ist folgender:

Bevor wir unsere neuen Assembler-Befehle ergänzt haben, stand an der Adresse \$1523 der Befehl RTS. Diesen haben wir jedoch durch den Befehl LDX #00 überschrieben (siehe oben)

Dieser Befehl gehört jedoch bereits zu dem Programmteil, welcher das Programm an die andere Stelle im Speicher kopiert. Wir müssen den Befehl RTS in der Kopie unseres Programms also manuell noch ergänzen nachdem der Kopiervorgang abgeschlossen wurde.

Dies geschieht durch die beiden Befehle

LDA #\$60
STA \$3023

Der erste Befehl lädt den Befehlscode \$60 (dezimal 96) in den Akkumulator, denn dies ist der Befehlscode für den Befehls RTS.

Der zweite Befehl schreibt den Inhalt des Akkumulators dann an die Adresse \$3023. Das ist exakt jene Adresse, an die der Befehl RTS in der Kopie unseres Programms landen muss (siehe übernächster Screenshot)

Nun speichern Sie das Programm mit dem Befehl

S“TELEPORTER“ 1500 1536

auf Diskette.

The screenshot shows a terminal window with assembly language code and a save command. The code consists of three lines:

```
;1530 H9 60 LDR #60
;1532 8D 23 30 STA 3023
;1535 60 RTS
```

Below the code is a dashed horizontal line, followed by the save command:

```
-----  
S"TELEPORTER" 1500 1536  
SAVING TELEPORTER  
.■
```

Nun wechseln Sie mit dem Befehl „X“ nach BASIC, löschen den Bildschirm, bewegen den Cursor einige Zeilen nach unten und starten das Programm mit SYS 5376.

Wie vorhin werden wieder die oberen und unteren 3 Zeilen mit den A's befüllt.

Nun prüfen wir, ob das Teleportieren an die Adresse 12288 erfolgreich war.

Löschen Sie nun den Bildschirm und geben SYS 12288 ein.

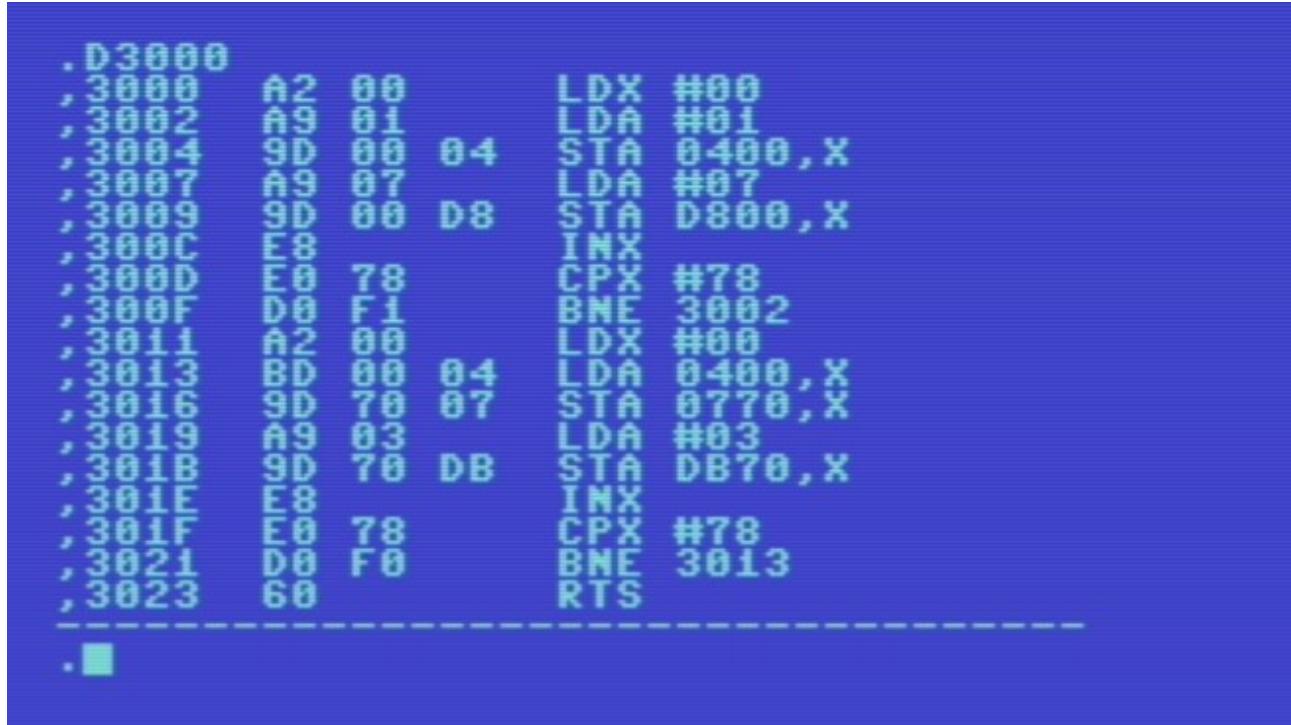
Sie sollten nun exakt das gleiche Ergebnis erhalten wie vorhin und die oberen und unteren 3 Zeilen sollten wiederum mit den gelben und türkisen A's befüllt werden.

Sehen wir uns doch auch mal an, wie die Kopie des Programms im Speicher aussieht, Wechseln Sie also zurück zu SMON und zeigen die Kopie des Programms mit dem Befehlen

D 3000

an.

Und tatsächlich liegt das Maschinenprogramm 1:1 als Kopie ab Adresse \$3000 im Speicher:



```
.D3000
,3000 A2 00 LDX #00
,3002 A9 01 LDA #01
,3004 9D 00 04 STA 0400,X
,3007 A9 07 LDA #07
,3009 9D 00 D8 STA D800,X
,300C E8 INX
,300D E0 78 CPX #78
,300F D0 F1 BNE 3002
,3011 A2 00 LDX #00
,3013 BD 00 04 LDA 0400,X
,3016 9D 70 07 STA 0770,X
,3019 A9 03 LDA #03
,301B 9D 70 DB STA DB70,X
,301E E8 INX
,301F E0 78 CPX #78
,3021 D0 F0 BNE 3013
,3023 60 RTS
```

Doch die Teleportation ist noch nicht abgeschlossen, denn das Programm existiert an seiner ursprünglichen Lage ja noch.

Nun kommt der zweite Schritt, das Programm soll seinen eigenen Maschinencode mit Nullwerten überschreiben (nur den Teil der die oberen unteren 3 Zeilen mit A's gefüllt und den Maschinencode an die andere Adresse kopiert)

Das Entfernen wird hier durch Überschreiben der Speicherstellen, welche den Programmcode beinhalten, mit dem Wert \$00, realisiert. Dies ist der Befehlscode für den Befehl BRK.

Doch nun zur Umsetzung.

An Adresse \$1535 steht aktuell der Befehl RTS.

Da wir unser Programm erneut erweitern wollen, müssen wir diesen überschreiben.

Durch den Befehl A 1535 werden die neuen Befehle ab der Adresse \$1535 gespeichert und daher der bisherige Befehl RTS überschrieben.

Geben Sie also die folgenden neuen Befehle ein und beenden die Eingabe wie gehabt mit dem Befehl „F“, sodass Sie folgende Anzeige erhalten:

```

,152B E8      INX
,152C E0 23   CPX #23
,152E D0 F5   BNE 1525
,153B A9 60   LDA #60
,1532 8D 23 30 STA 3023
,1535 60      RTS
-----
.A 1535
1535 A2 00   LDX #00
1537 A9 00   LDA #00
1539 9D 00 15 STA 1500,X
153C E8      INX
153D E0 39   CPX #39
153F D0 F8   BNE 1539
1541 60      RTS
1542 F
,1535 A2 00   LDX #00
,1537 A9 00   LDA #00
,1539 9D 00 15 STA 1500,X
,153C E8      INX
,153D E0 39   CPX #39
,153F D0 F8   BNE 1539
,1541 60      RTS
-----
.■

```

Nun speichern Sie das Programm am besten gleich auf Diskette ab:

```

,153C E8      INX
,153D E0 39   CPX #39
,153F D0 F8   BNE 1539
,1541 60      RTS
-----
S"TELEPORTER2" 1500 1542
SAVING TELEPORTER2
.■

```

Wechseln wir zurück zu Basic, starten das Programm mit SYS 5376 und wechseln gleich danach wieder zurück in den SMON.

Ob sich das Programm nun tatsächlich an der ursprünglichen Stelle in Luft aufgelöst hat, können wir ganz einfach durch Disassemblieren ab der Adresse \$1500 feststellen.

Hier die ersten 10 Bytes des Maschinencodes, welcher nun ja durch den Wert \$00 überschriebenen wurden.

```
.D 1500  
.1500 00      BRK  
.1501 00      BRK  
.1502 00      BRK  
.1503 00      BRK  
.1504 00      BRK  
.1505 00      BRK  
.1506 00      BRK  
.1507 00      BRK  
.1508 00      BRK  
.1509 00      BRK
```

Und hier das Ende des Programms. Wie man sieht, wurde alles bis auf die Schleife, welche das Überschreiben durchführt, überschrieben.

```
,1530 00      BRK  
.1531 00      BRK  
.1532 00      BRK  
.1533 00      BRK  
.1534 00      BRK  
.1535 00      BRK  
.1536 00      BRK  
.1537 00      BRK  
.1538 00      BRK  
.1539 9D 00 15  STA 1500,X  
.153C E8      INX  
.153D E0 39    CPX #39  
.153F D0 F8    BNE 1539  
.1541 60      RTS
```

Eine alternative Möglichkeit die Überschreibung zu visualisieren, besteht in einem Speicherdump mit dem SMON-Befehl „M“, den wir bereits kennengelernt haben.

M 1500 1542

```
. M 1500 1542
:1500 00 00 00 00 00 00 00 00 00
:1508 00 00 00 00 00 00 00 00 00
:1510 00 00 00 00 00 00 00 00 00
:1518 00 00 00 00 00 00 00 00 00
:1520 00 00 00 00 00 00 00 00 00
:1528 00 00 00 00 00 00 00 00 00
:1530 00 00 00 00 00 00 00 00 00
:1538 00 9D 00 15 E8 E0 39 D0
:1540 F8 60 FF FF FF FF 00 00
.■
```

Hier sieht man, dass von Adresse \$1500 (also der ursprünglichen Startadresse des Programms) bis hin zur Adresse \$1538 alle Speicherstellen den Wert \$00 beinhalten.

Ab Adresse \$1539 (Befehlscode \$9D für STA geht's dann los mit dem Maschinencode der Löschsleife und an Adresse \$1541 folgt schließlich das Ende des Programms durch den Befehl RTS, welches durch den Maschinencode \$60 (dezimal 96) erkennbar ist.

So, ich hoffe, ich habe Ihnen jetzt nicht den Spaß an der Assembler-Programmierung verdorben und dass Sie im Gegenteil noch immer mit großem Interesse dabei sind.

Glauben Sie mir, es lohnt sich! :)

4 Der Stapelspeicher und Unterprogramme

In diesem Kapitel werden wir uns mit Unterprogrammen beschäftigen. Vielleicht haben Sie in BASIC bereits für immer wiederkehrende Aufgaben Unterprogramme definiert und haben diese dann über den Befehl GOSUB aufgerufen.

Ähnlich funktioniert das auch in Assembler, doch bevor wir uns damit beschäftigen, müssen wir uns zuerst mit dem sogenannten Stapelspeicher, auch Stack genannt, befassen, da dieser in Bezug auf Unterprogramme eine wichtige Rolle spielt.

Der Stack erstreckt sich von Speicheradresse \$0100 (dezimal 256) bis Speicheradresse \$01FF (dezimal 511), ist also in Page Nr. 1 angesiedelt. Sie erinnern sich, eine Page ist ein Speicherbereich mit einem Umfang von 256 Bytes, von denen die erste im Speicher Zeropage genannt wird.

4.1 Was ist der Sinn und Zweck des Stacks?

Sie können sich den Stack als einen Ablageort für Informationen vorstellen, die man sich kurzfristig irgendwo merken muss. Auch die CPU nutzt den Stack zu diesem Zweck.

Wie der Name schon sagt, ist er wie ein Stapel organisiert, d.h. man legt entweder etwas oben drauf oder nimmt etwas von oben runter.

Das was man zuletzt draufgelegt hat, nimmt man auch als nächstes wieder runter. Dies nennt man auch das LIFO Prinzip (Last In First Out).

Das Gegenteil wäre das FIFO Prinzip (First In First Out).

Die CPU des C64 verfügt über einige Befehle, um auf den Stack zuzugreifen.

4.2 Befehle zum Zugriff auf den Stack

PHA (Push Accumulator, Befehlscode \$48)

Dieser Befehl legt den Inhalt des Akkumulator auf dem Stack ab.

PLA (Pull Accumulator, Befehlscode \$68)

Dieser Befehl holt sich den obersten Wert vom Stack und überträgt diesen in den Akkumulator.

PHP (Push Processor Status, Befehlscode \$08)

Dieser Befehl legt den Inhalt des Statusregisters auf dem Stack ab.

PLP (Pull Processor Status, Befehlscode \$28)

Dieser Befehl holt sich den obersten Wert vom Stack und überträgt diesen in das Statusregister.

TSX (Transfer Stackpointer to X, Befehlscode \$BA)

Dieser Befehl kopiert den Inhalt des SP Registers in das X Register.

TXS (Transfer X to Stackpointer, Befehlscode \$9A)

Dieser Befehl kopiert im umgekehrten Weg den Inhalt des X Registers in das SP Register.

Doch woher weiß die CPU, aus welcher Speicheradresse sie den Wert lesen muss, wenn wir ihr beispielsweise den Befehl PLA oder PLP geben?

Und woher weiß die CPU umgekehrt, an welche Speicheradresse sie einen Wert schreiben muss, wenn wir ihr den Befehl PHA oder PHP geben?

Hier kommt das SP Register (Stack Pointer Register) ins Spiel.

Es enthält immer die Position innerhalb des Stacks, an der der nächste Wert entweder mit dem Befehl PHA oder PHP abgelegt wird. Diese Position bewegt sich zwischen 0 und 255, da es ja als 8 Bit Register nur Werte aus diesem Wertebereich aufnehmen kann.

Wird also ein Wert auf den Stack gelegt, dann landet er also an der Position innerhalb des Stacks, welche aktuell im SP Register vermerkt ist.

Jetzt könnte man fragen:

OK, das SP Register enthält die aktuelle Position innerhalb des Stacks, also einen Wert zwischen 0 und 255. Woraus ergibt sich für die CPU dann aber die physikalische Speicheradresse für die Zugriffe auf den Stack?

Antwort:

Da der Stack ja fix bei Adresse \$0100 (dezimal 256) beginnt, ist es kein Problem wenn im SP Register nur eine Position steht. Die CPU addiert einfach die Position zur Startadresse \$0100 hinzu und erhält somit die benötigte Speicheradresse.

Nachdem der Wert dort abgelegt wurde, wird der Inhalt des SP Registers um 1 vermindert, denn an dieser Stelle landet dann der nächste Wert, den man auf den Stapel legen will.

Würde das SP Register nicht um 1 vermindert werden, dann würde der nächste Wert, den man auf den Stapel legt, ja wieder genau an derselben Stelle landen.

Wieso eigentlich vermindert? Müsste die Position nicht um 1 erhöht werden?

Das hat schon seine Richtigkeit, denn eine Besonderheit des Stacks ist, dass mit der Befüllung nicht an der ersten, sondern an der letzten Stelle begonnen wird und dieser in Richtung des Anfangs wächst und nicht umgekehrt.

Soll also ein Wert vom Stack genommen werden, so wird zuerst der Inhalt des SP Registers um 1 erhöht, denn die aktuelle Position ist ja jene, an die ein neuer Wert kommen würde und nicht jene, an der zuvor der letzte Wert abgelegt wurde.

Starten wir mal den C64 neu und laden den SMON wie gehabt.

Hinweis:

Wenn ich in den nachfolgenden Ausführungen den Begriff Stackpointer verwende, ist damit der Inhalt des SP Registers gemeint und umgekehrt. Ich meine also dasselbe.

The screenshot shows the Commodore 64 BASIC V2 interface. It displays the following text:
***** COMMODORE 64 BASIC V2 *****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
LOAD"SMONPC000",8,1
SEARCHING FOR SMONPC000
LOADING
READY.
NEW
READY.
SYS 49152

PC SR AC XR YR SP MU-BDIZC
;C00B B0 C2 00 00 F6 10110000
.■

Hier sehen wir, dass das SP Register den Wert \$F6 (dezimal 246) enthält, d.h. der nächste Wert den wir auf den Stack legen wollen kommt an die Position 246.

4.3 Beispiel-Programme

Geben Sie nun ab Adresse \$1500 folgende Befehle ein:

```
PC  SR  AC  XR  YR  SP    NU-BDIZC
;C00B B0 C2 00 00 F6    10110000
.A 1500
1500 A9 0A      LDA #0A
1502 48          PHA
1503 A9 14      LDA #14
1505 48          PHA
1506 A9 1E      LDA #1E
1508 48          PHA
1509 A9 28      LDA #28
150B 48          PHA
150C 00          BRK
150D F
,1500 A9 0A      LDA #0A
,1502 48          PHA
,1503 A9 14      LDA #14
,1505 48          PHA
,1506 A9 1E      LDA #1E
,1508 48          PHA
,1509 A9 28      LDA #28
,150B 48          PHA
,150C 00          BRK
-----.
■
```

Speichern Sie das Programm unter dem Namen „STACK1“ auf Diskette.

```
.S"STACK1" 1500 150D
.SAVING STACK1
■
```

Im ersten Befehl wird der Akkumulator mit dem Wert \$0A (dezimal 10) geladen und im nächsten Befehl wird dieser Wert auf den Stack gelegt.

Dann wird der Akkumulator mit dem Wert \$14 (dezimal 20) geladen und dieser ebenfalls auf den Stack gelegt.

Das wiederholt sich für die Werte \$1E (dezimal 30) und \$28 (dezimal 40)

Folgende Tabelle zeigt den Inhalt des SP Registers bevor der Wert auf den Stack gelegt wurde und nachdem der Wert auf den Stack gelegt wurde.

| SP Register vor dem Befehl PHA | Inhalt im Akkumulator | SP Register nach dem Befehl PHA |
|--------------------------------|-----------------------|---------------------------------|
| 246 (\$F6) | 10 (\$0A) | 245 (\$F5) |

| SP Register vor dem Befehl PHA | Inhalt im Akkumulator | SP Register nach dem Befehl PHA |
|--------------------------------|-----------------------|---------------------------------|
| 245 (\$F5) | 20 (\$14) | 244 (\$F4) |
| 244 (\$F4) | 30 (\$1E) | 243 (\$F3) |
| 243 (\$F3) | 40 (\$28) | 242 (\$F2) |

Wir sehen, dass sich mit jeder Ausführung des Befehls PHA der Inhalt des SP Registers um 1 vermindert.

Starten wir das Programm doch mal und sehen uns an, was auf dem Stack passiert ist.

```
.G 1500
PC  SR  AC  XR  YR  SP  NU-BDIZC
;1500D 30 28 00 00 F2 00110000
.■
```

Durch den folgenden Befehl können wir uns den Inhalt des Stacks anzeigen lassen:

```
.M 0100 01FF■
.■
:01C0 00 00 FF FF FF 00 00
:01C8 00 00 FF FF FF 00 00
:01D0 00 00 FF FF 7D EA FF 09
:01D8 7D EA 17 11 E2 E9 09 EA
:01E0 15 31 00 B8 E6 B8 E6 A7
:01E8 B8 E6 A7 E6 24 00 20 0A
:01F0 C4 11 C3 28 1E 14 0A 46
:01F8 E1 E9 A7 A7 79 A6 9C E3
.■
```

Hier sehen wir an der Adresse \$01F6 den Wert \$0A, also den ersten Wert, den wir auf den Stapel gelegt haben.

An der Adresse \$01F5 folgt der Wert \$14, also der zweite Wert, den wir auf den Stapel gelegt haben.

Anschließend folgen an der Adresse \$01F4 der dritte Wert \$1E und an der Adresse \$01F3 schließlich der vierte Wert \$28.

Das SP Register hat nach dem Ablegen des vierten Wertes den Inhalt \$F2, d.h. der nächste Wert, den wir auf den Stapel legen, würde hier links neben dem Wert \$28 an der Adresse \$01F2 landen.

Nun wollen wir diese 4 Werte wieder vom Stack nehmen. Ergänzen Sie also folgende Befehle ab der Adresse \$150C

```

.A 150C
,150C 68 PLA
,150D 68 PLA
,150E 68 PLA
,150F 68 PLA
,1510 00 BRK
,1511 F
,150C 68 PLA
,150D 68 PLA
,150E 68 PLA
,150F 68 PLA
,1510 00 BRK
-----
.
```

Das komplette Programm sollte dann so aussehen:

```

.D 1500
,1500 A9 0A LDA #0A
,1502 48 PHA
,1503 A9 14 LDA #14
,1505 48 PHA
,1506 A9 1E LDA #1E
,1508 48 PHA
,1509 A9 28 LDA #28
,150B 48 PHA
,150C 68 PLA
,150D 68 PLA
,150E 68 PLA
,150F 68 PLA
,1510 00 BRK
-----
.
```

Speichern Sie es mit dem Befehl S“STACK2“ 1500 1511 auf Diskette.

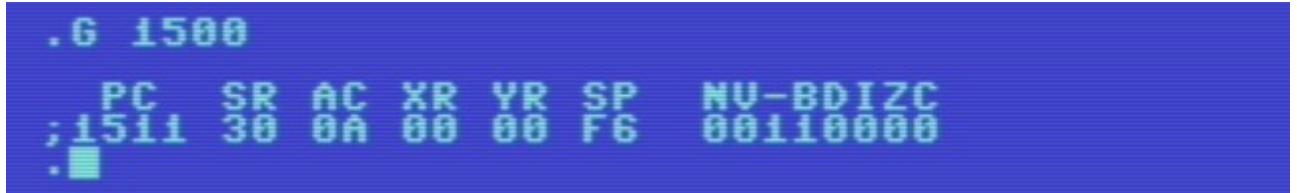
| SP Register vor dem Befehl PLA | SP Register nach dem Befehl PLA | Inhalt des Akkumulators nach dem Befehl PLA |
|--------------------------------|---------------------------------|---|
| 242 (\$F2) | 243 (\$F3) | 40 (\$28) |
| 243 (\$F3) | 244 (\$F4) | 30 (\$1E) |
| 244 (\$F4) | 245 (\$F5) | 20 (\$14) |
| 245 (\$F5) | 246 (\$F6) | 10 (\$0A) |

Hier wird insgesamt viermal der Befehl PLA aufgerufen. Durch den ersten Aufruf wird der Inhalt des SP Registers um 1 erhöht und der Wert \$28 vom Stack in den Akkumulator geholt, d.h. das SP Register enthält danach den Wert \$F3.

Durch den zweiten Aufruf wird das SP Register erneut um 1 erhöht und der Wert \$1E vom Stack in den Akkumulator geholt, d.h. das SP Register enthält danach den Wert \$F4. Der nächste Aufruf erhöht den Inhalt des SP Registers auf \$F5 und holt den Wert \$14 vom Stack in den Akkumulator.

Und der letzte Aufruf schließlich, erhöht den Inhalt des SP Registers auf \$F6 und holt den Wert \$0A vom Stack in den Akkumulator.

Wenn das Programm durchgelaufen ist, sehen wir, dass der Inhalt des SP Registers nun wieder bei \$F6 (dezimal 246) angekommen ist, also genau bei dem Inhalt, den es hatte, bevor wir den ersten Wert \$0A auf den Stack gelegt haben.



Das ist auch korrekt so, denn wir haben ja vier Werte auf den Stack gelegt und diese dann wieder vom Stack genommen.

Hier nochmal Schritt für Schritt was hier auf dem Stack passiert:

Ausgangssituation:

Noch keine Werte auf den Stack gelegt, Inhalt des SP Registers lautet \$F6 (dezimal 246), Inhalt an dieser Stelle unbekannt.

Schritt 1

LDA #\$0A
PHA

Der Wert \$0A (dezimal 10) wird auf den Stack gelegt und der Inhalt des SP Registers um 1 vermindert, enthält also nun den Wert \$F5 (dezimal 245)

Inhalt des Stacks:

| | Position | Inhalt |
|------|------------|-----------|
| | 246 (\$F6) | 10 (\$0A) |
| SP → | 245 (\$F5) | unbekannt |

Schritt 2

LDA #\$14
PHA

Der Wert \$14 (dezimal 20) wird auf den Stack gelegt und der Inhalt des SP Registers um 1 vermindert, enthält also nun den Wert \$F4 (dezimal 244)

Inhalt des Stacks:

| | Position | Inhalt |
|------|-----------------|---------------|
| | 246 (\$F6) | 10 (\$0A) |
| | 245 (\$F5) | 20 (\$14) |
| SP → | 244 (\$F4) | unbekannt |

Schritt 3

LDA #\$1E
PHA

Der Wert \$1E (dezimal 30) wird auf den Stack gelegt und der Inhalt des SP Registers um 1 vermindert, enthält also nun den Wert \$F3 (dezimal 243)

Inhalt des Stacks:

| | Position | Inhalt |
|------|-----------------|---------------|
| | 246 (\$F6) | 10 (\$0A) |
| | 245 (\$F5) | 20 (\$14) |
| | 244 (\$F4) | 30 (\$1E) |
| SP → | 243 (\$F3) | unbekannt |

Schritt 4

LDA #\$28
PHA

Der Wert \$28 (dezimal 40) wird auf den Stack gelegt und der Inhalt des SP Registers um 1 vermindert, enthält also nun den Wert \$F2 (dezimal 242)

Inhalt des Stacks:

| | Position | Inhalt |
|------|-----------------|---------------|
| | 246 (\$F6) | 10 (\$0A) |
| | 245 (\$F5) | 20 (\$14) |
| | 244 (\$F4) | 30 (\$1E) |
| | 243 (\$F3) | 40 (\$28) |
| SP → | 242 (\$F2) | unbekannt |

Genereller Hinweis:

Durch den Aufruf von PLA wird zwar der oberste Wert vom Stapel in den Akkumulator kopiert, der Wert bleibt jedoch grundsätzlich im Speicher stehen und es findet in diesem Sinne also keine

"Entfernung" statt, so wie es bei einem Stapel Bücher der Fall sein würde, wenn man das oberste Buch vom Stapel nimmt.

Der Wert, den man durch PLA "vom Stapel genommen" hat, befindet sich jedoch nun im freien Bereich des Stacks, d.h. er wird durch den nächsten Wert, der auf den Stack gelegt wird, überschrieben.

Und da außer unserem Programm ja noch andere Programme auf den Stack zugreifen (z.B. Betriebssystem-Routinen), wird diese Überschreibung nicht lange auf sich warten lassen.

Die Werte, die ich nachfolgend bei den Schritten 5 – 8 in der Spalte „Inhalt“ angegeben habe, wären also nur gültig, wenn nach der Ausführung von PLA noch kein anderes Programm Veränderungen auf dem Stack vorgenommen hat.

Allein der Inhalt des SP Registers bestimmt die aktuelle "Höhe" des Stacks.

Schritt 5

PLA

Der Inhalt des SP Registers wird um 1 erhöht, es enthält also nun den Wert \$F3 (dezimal 243), dann wird der Wert an dieser Position ausgelesen und in den Akkumulator geschrieben.

Inhalt des Stacks:

| | Position | Inhalt |
|------|-----------------|---------------|
| | 246 (\$F6) | 10 (\$0A) |
| | 245 (\$F5) | 20 (\$14) |
| | 244 (\$F4) | 30 (\$1E) |
| SP → | 243 (\$F3) | 40 (\$28) |
| | 242 (\$F2) | unbekannt |

Schritt 6

PLA

Der Inhalt des SP Registers wird um 1 erhöht, es enthält also nun den Wert \$F4 (dezimal 244), dann wird der Wert an dieser Position ausgelesen und in den Akkumulator geschrieben.

Inhalt des Stacks:

| | Position | Inhalt |
|------|-----------------|---------------|
| | 246 (\$F6) | 10 (\$0A) |
| | 245 (\$F5) | 20 (\$14) |
| SP → | 244 (\$F4) | 30 (\$1E) |
| | 243 (\$F3) | 40 (\$28) |
| | 242 (\$F2) | unbekannt |

Schritt 7

PLA

Der Inhalt des SP Registers wird um 1 erhöht, es enthält also nun den Wert \$F5 (dezimal 245), dann wird der Wert an dieser Position ausgelesen und in den Akkumulator geschrieben.

Inhalt des Stacks:

| | Position | Inhalt |
|------|-----------------|---------------|
| | 246 (\$F6) | 10 (\$0A) |
| SP → | 245 (\$F5) | 20 (\$14) |
| | 244 (\$F4) | 30 (\$1E) |
| | 243 (\$F3) | 40 (\$28) |
| | 242 (\$F2) | unbekannt |

Schritt 8

PLA

Der Inhalt des SP Registers wird um 1 erhöht, es enthält also nun den Wert \$F6 (dezimal 246), dann wird der Wert an dieser Position ausgelesen und in den Akkumulator geschrieben.

Inhalt des Stacks:

| | Position | Inhalt |
|------|-----------------|---------------|
| SP → | 246 (\$F6) | 10 (\$0A) |
| | 245 (\$F5) | 20 (\$14) |
| | 244 (\$F4) | 30 (\$1E) |
| | 243 (\$F3) | 40 (\$28) |
| | 242 (\$F2) | unbekannt |

Damit ist der Inhalt der SP Registers wieder dort angekommen, wo er war, bevor wir den ersten Wert auf den Stack gelegt haben.

Angenommen, man würde nun den Wert \$C8 (dezimal 200) auf den Stack legen:

```
LDA #$C8  
PHA
```

Dann würde der Stack so aussehen:

| | Position | Inhalt |
|------|-----------------|---------------|
| | 246 (\$F6) | 200 (\$C8) |
| SP → | 245 (\$F5) | 20 (\$14) |
| | 244 (\$F4) | 30 (\$1E) |
| | 243 (\$F3) | 40 (\$28) |

Fassen wir also alles nochmal zusammen:

Der Stack liegt im Speicherbereich von \$0100 bis \$01FF, umfasst also die Page 1 im Speicher.

Wenn wir einen Wert auf den Stapel legen (durch PHA oder PHP), dann wird entweder der Inhalt des Akkumulators (im Fall von PHA) oder der Inhalt des Statusregisters (im Fall von PHP) an die Position auf dem Stack abgelegt auf die das SP Register verweist. Anschließend wird der Inhalt des SP Registers um 1 vermindert.

Wenn wir einen Wert vom Stapel holen (durch PLA oder PLP), dann wird zunächst der Inhalt des SP Registers um 1 erhöht und der Wert an dieser Position des Stacks gelesen. Dieser Wert wird dann in das jeweilige Zielregister (Akkumulator im Fall von PLA oder Statusregister im Fall von PLP) geschrieben.

Der Stack wird in umgekehrter Richtung (beginnend am Ende in Richtung Anfang) befüllt.

5 Unterprogramme

Um den Sinn und Zweck von Unterprogrammen zu erklären, möchte ich mich auf das Beispiel aus dem letzten Kapitel beziehen, in dem es darum ging, die ersten drei Zeilen des Bildschirms mit einem A in gelber Farbe zu füllen.

Dazu haben wir folgenden Assembler-Code geschrieben:

The screenshot shows a text-based assembly language editor. The code is as follows:

```
.D 1500
,1500 A2 00      LDX #00
,1502 A9 01      LDA #01
,1504 9D 00 04    STA 0400,X
,1507 A9 07      LDA #07
,1509 9D 00 D8    STA D800,X
,150C E8          INX
,150D E0 78      CPX #78
,150F D0 F1      BNE 1502
,1511 60          RTS
-----
.■
```

Angenommen, wir schreiben ein größeres Assembler-Programm und möchten an mehreren Stellen in diesem Programm die obersten drei Zeilen mit dem gelben A füllen. Dann müssten wir an all diesen Stellen den obigen Assembler-Code in unser Programm schreiben. Das würde erstens viel Schreibaufwand bedeuten und zweitens die Fehleranfälligkeit erhöhen.

Da wäre es doch sehr angenehm, wenn wir den Code nur einmal schreiben müssten und dann beliebig oft, von jeder nur denkbaren Stelle in unserem Programm, aufrufen könnten, so wie wir es von BASIC mit GOSUB kennen.

Und natürlich gibt es diese Möglichkeit.

Durch den Befehl JSR (Jump to Subroutine) haben wir die Möglichkeit, die Programmausführung an einer anderen Adresse (welche wir dem Befehl JSR als Parameter übergeben), fortzusetzen und dann mit dem Befehl RTS wieder zu der Stelle zurückzukehren, an der wir den Aufruf durchgeführt haben.

Die Programmausführung wird dann mit dem Befehl fortgesetzt der dem Befehl JSR folgt.

Der Befehl JSR ist also quasi das Assembler-Gegenstück zu dem BASIC-Befehl GOSUB und der Befehl RTS hat in diesem Zusammenhang dieselbe Funktion wie der BASIC-Befehl RETURN.

Dieser bewirkt ebenfalls eine Fortsetzung des Programms mit der Anweisung, welche dem Befehl GOSUB folgt.

Den Code von Adresse \$1500 bis \$1511 verändern wir nicht, denn diesen können wir nun als Code für unser Unterprogramm ansehen.

Was wir nun noch ergänzen müssen, ist der Code, der das Unterprogramm an Adresse \$1500 aufruft:

```
.A 1512
1512 20 00 15 JSR 1500
1515 60 RTS
1516 F
,1512 20 00 15 JSR 1500
,1515 60 RTS
-----
.■
```

Das komplette Programm müsste nun so aussehen:

```
.D 1500
,1500 A2 00 LDX #00
,1502 A9 01 LDA #01
,1504 9D 00 04 STA $400,X
,1507 A9 07 LDA #07
,1509 9D 00 D8 STA $800,X
,150C E8 INX
,150D E0 78 CPX #78
,150F D0 F1 BNE 1502
,1511 60 RTS
-----
,1512 20 00 15 JSR 1500
,1515 60 RTS
-----
.■
```

Wie funktioniert das Programm?

Der Code von Adresse \$1500 bis \$1511 stellt unser Unterprogramm dar.

Der Befehl JSR \$1500 bewirkt, dass die Programmausführung an der Programmadresse \$1500 fortgesetzt wird.

Der Befehl RTS an der Programmadresse \$1511 bewirkt dann den Rücksprung aus dem Unterprogramm und die Programmausführung wird an der Programmadresse \$1515 fortgesetzt.

In diesem Fall ist das der Befehl RTS, wodurch unser Programm beendet wird.

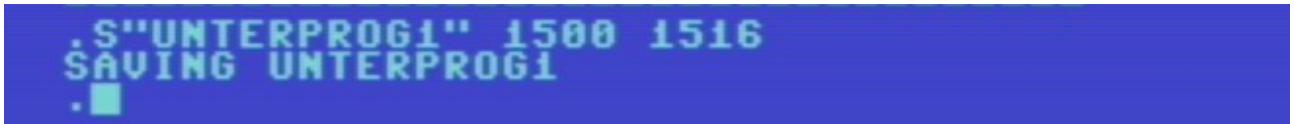
Wechseln wir mit „X“ zurück nach Basic, löschen den Bildschirm und bewegen den Cursor ein paar Zeilen nach unten.

Da unser Programm ja nun an der Adresse \$1512 (dezimal 5394) beginnt, müssen wir das Programm mit SYS 5394 starten.



Das Ergebnis ist dasselbe wie vorhin ohne Unterprogramm, doch wir können diesen Programmteil nun beliebig oft, von jeder beliebigen Stelle im Programm aus, mit JSR aufrufen und ersparen uns das unnötige und fehleranfällige Vervielfachen dieses Programmcodes.

Speichern Sie das Programm unter dem Namen „UNTERPROG1“ ab.



Doch welche Schritte werden von der CPU bei einem solchen Sprung in ein Unterprogramm durchgeführt?

Hier kommt nun der Stack ins Spiel.

Der Befehl JSR ist ein Befehl, welcher 3 Bytes beansprucht (1 Byte für den Befehlscode + 1 Byte für das niederwertige Byte der Zieladresse + 1 Byte für das höherwertige Byte der Zieladresse)

Als ersten Schritt erhöht die CPU den Inhalt des Program Counter Registers um 2 (Länge des JSR Befehls - 1, 3 - 1 = 2)

Als nächstes legt sie das höherwertige Byte des neuen Inhalts des Program Counter Registers auf den Stack, wodurch sich der Inhalt des SP Registers um 1 vermindert.

Dann legt sie das niederwertige Byte des neuen Inhalts des Program Counter Registers auf den Stack, wodurch sich der Inhalt des SP Registers wiederum um 1 vermindert.

Schließlich trägt sie die Zieladresse, welche wir dem JSR Befehl als Parameter angegeben haben, in das Program Counter Register ein, wodurch das Programm an dieser Adresse fortgesetzt wird.

Wenn im Unterprogramm der Befehl RTS erreicht wird, führt sie umgekehrt folgende Schritte durch:

Sie erhöht den Inhalt des SP Registers um 1 und liest den Wert an dieser Position vom Stack (dies ist der niederwertige Teil der Adresse, den sie zuvor auf den Stack gelegt hat)

Diesen Wert trägt sie in den niederwertigen Teil des Program Counter Registers ein.

Dann erhöht sie wiederum den Inhalt des SP Registers um 1 und liest den Wert an dieser Position vom Stack (dies ist der höherwertige Teil der Adresse, den sie zuvor auf den Stack gelegt hat)

Diesen Wert trägt sie in den höherwertigen Teil des Program Counter Registers ein.

Danach addiert sie noch 1 zum Inhalt des Program Counter Registers, wodurch die Programmausführung mit jenem Befehl fortgesetzt wird, der dem Befehl JSR folgt.

Soweit so gut, doch unser Unterprogramm hat ein Problem.

Wir können es zwar beliebig oft aufrufen, aber es wird immer nur die obersten 3 Zeilen mit einem gelben A füllen. Wollen wir stattdessen grüne B's anzeigen, dann funktioniert das nicht, weil im Unterprogramm fix festgelegt ist, dass gelbe A's angezeigt werden sollen.

LDA #\$01 <- hier wird der Zeichencode fix festgelegt (in diesem Fall 1 für den Buchstaben A)
STA \$0400,X

LDA #\$07 <- hier wird der Farocode fix festgelegt (in diesem Fall 7 für die Farbe Gelb)
STA \$D800,X

Da wäre es doch schön, wenn wir dem Unterprogramm sagen könnten, mit welchem Zeichen wir die Zeilen füllen wollen und in welcher Farbe die Zeichen dargestellt werden sollen.

Dann könnten wir bei einem Aufruf des Unterprogramms die Zeilen mit einem gelben A füllen, ein anderes mal mit einem türkisen C und wieder ein anderes mal mit einem schwarzen Y zum Beispiel.

Auch Parameter zur Angabe der Anzahl der Zeilen bzw. mit welcher Zeile begonnen wird, wäre denkbar. Wir wollen uns für das folgende Beispiel jedoch auf das auszugebende Zeichen und die Farbe beschränken.

Doch wie geben wir diese Informationen an unser Unterprogramm weiter?

Hier stehen mehrere Möglichkeiten zur Verfügung, die alle ihr Vor- und Nachteile haben.

5.1 Übergabe von Parametern an Unterprogramme

5.1.1 Übergabe der Parameter in CPU Registern

Diese Methode wird vor allem dann genutzt, wenn das Unterprogramm nur bis zu drei Parameter benötigt. Vor dem Aufruf schreibt man die Parameter je nach Anzahl in den Akkumulator, das X Register oder das Y Register und ruft das Unterprogramm auf.

Das Unterprogramm liest die jeweiligen Parameter dann aus den Registern aus und verwendet sie je nachdem wie sich die Aufgabe des Unterprogramms gestaltet.

Rein von der Zugriffsgeschwindigkeit aus betrachtet, liegt diese Methode auf Platz eins, da die Parameter ja direkt in den Registern liegen und kein Umweg über den Hauptspeicher nötig ist.

In unserem Beispiel werden wir den Zeichencode im Akkumulator und den Farbcode im Y Register an das Unterprogramm übergeben. Das X Register verwenden wir ja bereits als Schleifenzähler.

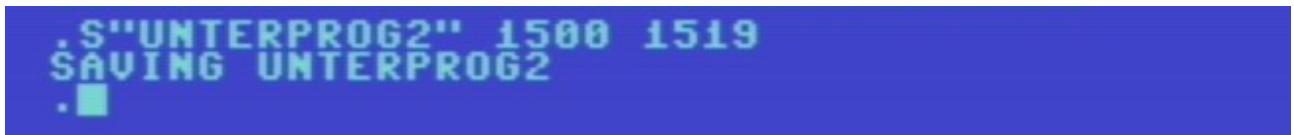
```
.A 1500
1500 A2 00      LDX #00
1502 48          PHA
1503 9D 00 04    STA 0400,X
1506 98          TYA
1507 9D 00 D8    STA D800,X
150A 68          PLA
150B E8          INX
150C E0 78      CPX #78
150E D0 F2      BNE 1502
1510 60          RTS
1511 A9 01      LDA #01
1513 A0 07      LDY #07
1515 20 00 15    JSR 1500
1518 60          RTS
1519 F■
```

Schließen Sie die Eingabe wie üblich mit „F“ ab.

```
150C E0 78      CPX #78
150E D0 F2      BNE 1502
1510 60          RTS
1511 A9 01      LDA #01
1513 A0 07      LDY #07
1515 20 00 15    JSR 1500
1518 60          RTS
1519 F
,1500 A2 00      LDX #00
,1502 48          PHA
,1503 9D 00 04    STA 0400,X
,1506 98          TYA
,1507 9D 00 D8    STA D800,X
,150A 68          PLA
,150B E8          INX
,150C E0 78      CPX #78
,150E D0 F2      BNE 1502
,1510 60          RTS
-----
,1511 A9 01      LDA #01
,1513 A0 07      LDY #07
,1515 20 00 15    JSR 1500
,1518 60          RTS
-----■
```

Der Code des Unterprogramms reicht von Adresse \$1500 bis Adresse \$1510

Ab Adresse \$1511 beginnt der Code, der unser Unterprogramm aufruft. In diesem Beispiel verwenden wir den Akkumulator und das Y Register zur Übergabe unserer Parameter. Speichern Sie das Programm unter dem Namen „UNTERPROG2“ mit dem Befehl



auf Diskette.

Doch nun zum Ablauf des Programms.

Im Akkumulator speichern wir den Zeichencode des Zeichens, mit dem wir die 3 obersten Zeilen befüllen wollen und im Y Register speichern wir den gewünschten Farbcode.

Um bei dem vorherigen Beispiel zu bleiben, habe ich hier ebenfalls den Zeichencode 1 (für den Buchstaben A) und den Farbcode 7 (für die Farbe Gelb) verwendet.

Nachdem wir diese beiden Register geladen haben, wird das Unterprogramm durch den Befehl JSR \$1500 aufgerufen.

Dort wird als Erstes wie vorhin das X Register mit 0 initialisiert, da es ja als unser Schleifenzähler für die 120 Durchläufe dient.

Durch den Befehl PHA sichern wir den Inhalt des Akkumulators (enthält den Zeichencode 1) auf dem Stack, da er durch den übernächsten Befehl TYA überschrieben wird.

Doch zuvor wird der Zeichencode 1 durch den Befehl STA \$0400,X in die aktuelle Speicherstelle im Videospeicher geschrieben.

Nun wird durch den Befehl TYA der Inhalt des Y Registers, welches ja den Farbcode enthält, in den Akkumulator übertragen.

Durch den Befehl STA \$D800,X wird nun der Farbcode für das zuvor ausgegebene A an der richtigen Speicherstelle im Farbspeicher eingetragen.

Dann holen wir durch den Befehl PLA den zuvor auf dem Stack gesicherten Wert wieder zurück in den Akkumulator, sodass dieser nun wieder den Zeichencode 1 enthält.

Der Rest läuft ab wie vorhin, der Inhalt des X Registers wird um 1 erhöht, mit dem Wert \$78 (dezimal 120) verglichen und solange dieser Wert nicht erreicht ist, wird zu der Programmadresse \$1502 gesprungen.

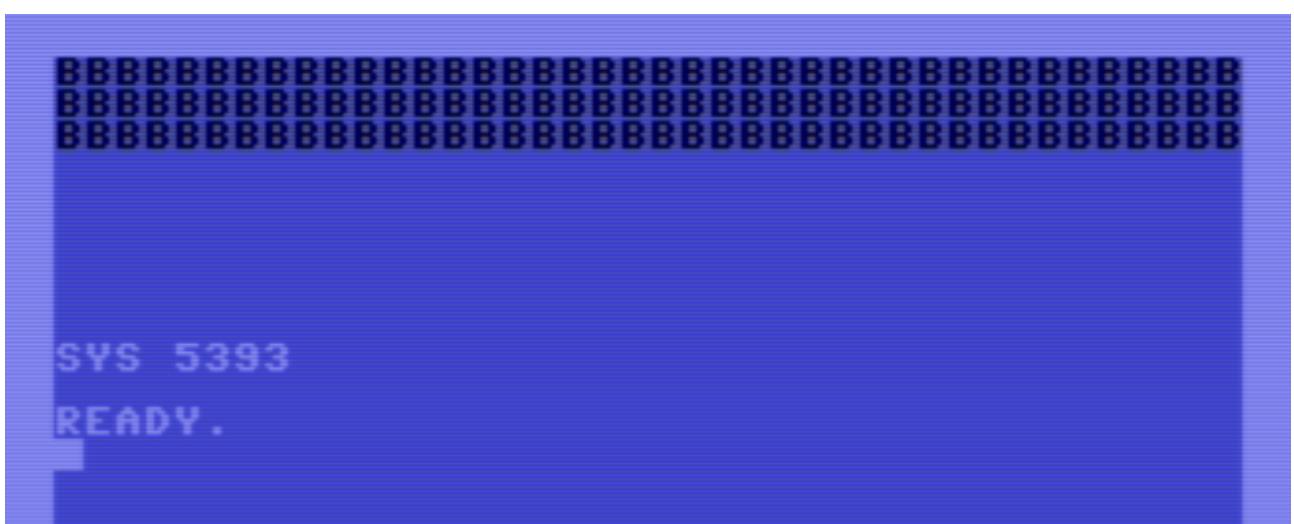
Ein A nach dem anderen wird ausgegeben, bis die obersten 3 Zeilen vollgeschrieben sind.



Ändern Sie nun den Befehl LDA an Adresse \$1511 und den Befehl LDY, sodass nun der Zeichencode 2 (für den Buchstaben B) und der Farbcode 0 (für die Farbe Schwarz) verwendet wird.



Wechseln Sie wie gehabt mit „X“ nach BASIC, löschen den Bildschirm, bewegen den Cursor um einige Zeilen nach unten und starten das Programm mit SYS 5393.



Nun werden statt den gelben A's schwarze B's angezeigt.

An unserem Unterprogramm haben wir nichts verändert, wir haben ausschließlich die Parameterwerte in den Registern geändert.

Wiederholen Sie nun den Vorgang mit Zeichencode 3 für den Buchstaben C und Farbcode 3 für die Farbe Türkis.

```
-----  
. 1511 A9 03      LDA #03  
. 1513 A0 03      LDY #03  
W 1515 20 00 15    JSR 1500  
. 1518 60          RTS  
-----  
.
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC  
  
SYS 5393  
READY.
```

Nun wollen wir mal mehrere Aufrufe des Unterprogramms kombinieren.

Geben Sie ab Adresse \$1511 folgende neue Befehle ein und schließen die Eingabe wie üblich mit „F“ ab.

```
.A 1511  
1511 A9 01      LDA #01  
1513 A0 07      LDY #07  
1515 20 00 15    JSR 1500  
1518 20 E4 FF    JSR FFE4  
151B F0 FB      BEQ 1518  
151D A9 02      LDA #02  
151F A0 00      LDY #00  
1521 20 00 15    JSR 1500  
1524 20 E4 FF    JSR FFE4  
1527 F0 FB      BEQ 1524  
1529 A9 03      LDA #03  
152B A0 03      LDY #03  
152D 20 00 15    JSR 1500  
1530 60          RTS  
1531 F■
```

Speichern Sie das Programm unter dem Namen „UNTERPROG3“ mit folgendem Befehl ab.

```
.S"UNTERPROG3" 1500 1531  
SAVING UNTERPROG3  
.■
```

Wechseln Sie zurück zu BASIC, löschen den Bildschirm, bewegen den Cursor um einige Zeilen nach unten und starten das Programm wiederum mit SYS 5393.

Es werden 3 Zeilen mit gelben A's angezeigt



Drücken Sie eine beliebige Taste, es werden 3 Zeilen mit schwarzen B's angezeigt.



Drücken Sie wiederum eine beliebige Taste, es werden 3 Zeilen mit türkisen C's angezeigt und das Programm beendet.



Hier haben wir die drei vorhergehenden Aufrufe des Unterprogramms kombiniert, allerdings musste ich zwischen den Schritten etwas einbauen, um das Programm anzuhalten, damit Sie das Ergebnis der jeweiligen Ausgabe sehen können.

Dies habe ich in Form einer Schleife realisiert, die solange durchlaufen wird, bis eine beliebige Taste gedrückt wird. Sobald eine Taste gedrückt wird, wird das Programm fortgesetzt.

Realisiert wird das durch folgende Zeilen:

bzw.

תְּמִימָנָה תְּמִימָנָה תְּמִימָנָה תְּמִימָנָה תְּמִימָנָה תְּמִימָנָה תְּמִימָנָה

Dies ist eigentlich ein Vorgriff auf das nächste Kapitel, in dem es um den Aufruf von den sogenannten KERNAL-Routinen geht.

Doch die Funktion ist relativ einfach erklärt.

Eingaben über die Tastatur werden zunächst im Tastaturpuffer abgelegt. Das ist ein relativ kleiner Speicherbereich, der dazu dient, die Codes jener Tasten, die der Benutzer gedrückt hat, zwischenzuspeichern.

Wir rufen hier über die Adresse \$FFE4 ein Unterprogramm auf, welches bereits vom Betriebssystem des C64 zur Verfügung gestellt wird. Es hat die Aufgabe, nachzusehen, ob im Tastaturpuffer ein Tastencode bereitsteht. Ist das der Fall, speichert es diesen Tastencode im Akkumulator ab und kehrt dann zum Aufrufer zurück.

Steht im Tastaturpuffer kein Tastencode zur Verarbeitung bereit, legt das Unterprogramm den Wert 0 im Akkumulator ab.

In diesem Fall wird das Zeroflag gesetzt. Warum? Weil es nicht nur dann gesetzt wird, wenn das Ergebnis einer Rechenoperation gleich 0 ist, sondern auch dann, wenn eine 0 in ein Register geladen wird.

Hier kommt das Gegenstück zu dem Befehl BNE zum Einsatz, nämlich der Befehl BEQ. Durch ihn wird wieder zur Programmadresse \$1518 bzw. bei der zweiten Schleife zur Programmadresse \$1524 gesprungen, was einen erneuten Aufruf des Unterprogramms bewirkt.

Das setzt sich solange fort, bis wir eine Taste drücken. Drücken wir eine Taste, wird der Tastencode der Taste, die wir gedrückt haben, in den Tastaturpuffer gelegt. Das oben genannte Unterprogramm stellt dann fest, dass im Tastaturpuffer ein Code zur Verarbeitung bereitsteht und liefert uns im Akkumulator diesen Code zurück, damit wir diesen weiterverarbeiten können.

Dann wird die Programmausführung mit dem Befehl fortgesetzt der auf den Befehl BEQ folgt.

Dadurch bleiben die gelben A's bzw. die schwarzen B's solange am Bildschirm sichtbar, bis Sie eine Taste drücken.

Würden wir diese Tastaturabfrage nicht einbauen, ginge alles so schnell, dass Sie den Wechsel gar nicht mitbekommen würden und Sie würden nur die türkisen C's sehen, weil das Assembler-Programm zu schnell durchläuft. Probieren Sie es gerne aus!

5.1.2 Übergabe der Parameter in ausgesuchten Stellen in der Zeropage

Wie bereits bekannt, wird der Speicherbereich von Adresse 0 bis 255 (\$00 - \$FF) als Zeropage bezeichnet. Dort werden viele wichtige Informationen aufbewahrt, die für den reibungslosen Betrieb des C64 wichtig sind. Trotzdem gibt es zwischendurch immer wieder vereinzelte Speicherstellen, die man beispielsweise als Übergabeort für Parameter verwenden kann.

Angenommen, man verwendet keine Datasette, dann könnte man jene Speicherstellen in der Zeropage nutzen, die Informationen für den Betrieb einer Datasette beinhalten.

Aus Sicht der Zugriffsgeschwindigkeit liegt diese Methode auf Platz zwei, da für die Adressen in diesem Bereich ja nur 1 Byte aus dem Speicher gelesen werden muss.

Diese Methode ist jedoch sehr mit Vorsicht zu genießen und man muss darauf achten, dass man keine Speicherstellen verwendet, die vom System anderweitig verwendet werden.

In unserem Beispiel hier werden wir die Speicherstelle \$FB (dezimal 251) dazu verwenden um den Zeichencode zu übergeben und die Speicherstelle \$FC um den Farocode zu übergeben.

Geben Sie folgendes Programm ab der Adresse \$1500 ein und schließen die Eingabe wie üblich mit „F“ ab.

```

.A 1500
1500 A2 00      LDX #00
1502 A5 FB      LDA FB
1504 9D 00 04    STA 0400,X
1507 A5 FC      LDA FC
1509 9D 00 D8    STA D800,X
150C E8          INX
150D E0 78      CPX #78
150F D0 F1      BNE 1502
1511 60          RTS
1512 20 00 15    JSR 1500
1515 60          RTS
1516 F■

```

Speichern Sie das Programm unter dem Namen „ZEROPAGEASM“

```

1515 60          RTS
1516 F■
,1500 A2 00      LDX #00
,1502 A5 FB      LDA FB
,1504 9D 00 04    STA 0400,X
,1507 A5 FC      LDA FC
,1509 9D 00 D8    STA D800,X
,150C E8          INX
,150D E0 78      CPX #78
,150F D0 F1      BNE 1502
,1511 60          RTS
-----
,1512 20 00 15    JSR 1500
,1515 60          RTS
-----
.S"ZEROPAGEASM" 1500 1516
SAVING ZEROPAGEASM
.■

```

auf Diskette.

Wie funktioniert das Unterprogramm?

Im ersten Befehl wird wieder das X Register als Schleifenzähler mit dem Startwert 0 initialisiert. Dann wird der erste Parameter (Zeichencode) aus der Speicherstelle \$FB (dezimal 251) gelesen und mit dem nächsten Befehl an die jeweilige Position im Videospeicher geschrieben.

Anschließend wird der Parameter für den Farbcode aus der Speicherstelle \$FC (dezimal 252) gelesen und an die jeweilige Position im Videospeicher geschrieben. Der Rest des Unterprogramms hat sich nicht verändert.

Ich werde Ihnen nun zeigen, wie ein BASIC-Programm und ein Maschinenprogramm zusammenarbeiten können.

Wechseln Sie zurück zu BASIC, geben folgendes Programm ein und speichern es unter dem Namen „ZEROPAGEBAS“.

The screenshot shows a Commodore 64 monitor displaying assembly language code. The code is as follows:

```
,1512 20 00 15 JSR 1500
,1515 60 RTS
-----
.S"ZEROPAGEASM" 1500 1516
SAVING ZEROPAGEASM
.X
READY.
10 POKE 251,1
20 POKE 252,7
30 SYS 5394
40 GET A$
50 IF A$="" THEN 40
60 POKE 251,2
70 POKE 252,0
80 SYS 5394
90 GET A$
100 IF A$="" THEN 90
110 POKE 251,3
120 POKE 252,3
130 SYS 5394
SAVE"ZEROPAGEBAS",8
SAVING ZEROPAGEBAS
READY.
```

Löschen Sie den Bildschirm, bewegen den Cursor einige Zeilen nach unten und starten das Programm mit RUN.

Das Ergebnis ist wiederum dasselbe, zuerst werden die gelben A's, dann die schwarzen B's und letztendlich die türkisen C's angezeigt.

Doch dieses mal haben wir kein reines Maschinenprogramm geschrieben, sondern ein BASIC-Programm, welches die Parameter für das Unterprogramm über die Speicherstellen 251 (\$FB) und 252 (\$FC) festlegt und dieses dann mit SYS 5394 aufruft.

Warum gerade 251 und 252? Die Speicherstellen 251-254 in der Zeropage stehen zur freien Verfügung, sofern sie nicht schon ein anderes Programm verwendet.

Die Tastenabfrage dazwischen können wir nun auch im BASIC-Programm erledigen und brauchen nicht den Weg über das Unterprogramm an der Speicheradresse \$FFE4 gehen.

5.1.3 Übergabe der Parameter in einem eigenen Datenbereich an passender Stelle im Hauptspeicher

Diese Methode ist eigentlich identisch zur vorherigen, der Unterschied liegt einzig und allein in den Speicherstellen, welche für die Parameter verwendet werden.

Bei der vorherigen Methode haben wir die Parameter in der Zeropage platziert, d.h. wir konnten sie über ein Adresse ansprechen, die nur ein Byte benötigt.

Was aber, wenn in sich in der Zeropage keine Möglichkeiten mehr zur Parameterablage bieten?

In diesem Fall kann man auf andere Speicherstellen im Hauptspeicher ausweichen, der Nachteil ist allerdings, dass man es mit Adressen > 255 zu tun hat, d.h. wir kommen bei der Adressierung nicht mehr mit einem Byte aus, sondern haben es immer mit 16 Bit Adressen zu tun.

Doch welche Speicherstellen verwendet man?

Im Grunde kann man alle Speicherstellen verwenden, die nicht bereits anderweitig in Verwendung sind. In diesem Beispiel hier habe ich am Anfang des Programms zwei Speicherstellen als Platzhalter eingebaut, welche ich für die Übergabe der Parameter verwenden will.

Vor dem Aufruf des Unterprogramms legt man dann die Parameter in diesen Speicherstellen ab und das Unterprogramm holt sich die Werte dann aus diesen Speicherstellen.

Da der Zugriff hier wie gesagt über 16 Bit Adressen stattfindet, ist diese Methode langsamer als die bisher genannten.

Ich werde auch hier eine Kombination aus BASIC-Programm und Maschinenprogramm verwenden.

Geben Sie folgende Befehle ab Adresse \$1500 ein und schließen die Eingabe wie üblich mit „F“ ab:

Hier lernen Sie auch einen neuen Assembler-Befehl kennen, nämlich den Befehl NOP (No Operation)

Die Funktion des Befehl ist leicht erklärt: Er tut rein gar nichts, er verbraucht nur Zeit und wird deshalb oft verwendet, um Wartezeiten im Programm einzulegen oder eben, so wie hier, um Platzhalter in das Programm einzubauen.

Ich verwende den Befehl hier jedoch nicht, weil ich Zeit verbrauchen will, sondern weil ich Ihnen zeigen möchte, welche beiden Speicherstellen ich für die Parameter verwende.

```
.A 1500
1500 EA      NOP
1501 EA      NOP
1502 A2 00    LDX #00
1504 AD 00 15  LDA 1500
1507 9D 00 04  STA 0400,X
150A AD 01 15  LDA 1501
150D 9D 00 D8  STA D800,X
1510 E8      INX
1511 E0 78    CPX #78
1513 D0 EF    BNE 1504
1515 60      RTS
1516 20 02 15  JSR 1502
1519 60      RTS
151A F■
```

Speichern Sie das Programm unter dem Namen „RAMPARAMASM“

```

1510 E8      INX
1511 E0 78    CPX #78
1513 D0 EF    BNE 1504
1515 60      RTS
1516 20 02 15 JSR 1502
1519 60      RTS
151A F
,1500 EA      NOP
,1501 EA      NOP
,1502 A2 00    LDX #00
,1504 AD 00 15  LDA 1500
,1507 9D 00 04  STA 0400,X
,150A AD 01 15  LDA 1501
,150D 9D 00 D8  STA D800,X
,1510 E8      INX
,1511 E0 78    CPX #78
,1513 D0 EF    BNE 1504
,1515 60      RTS
-----
,1516 20 02 15 JSR 1502
,1519 60      RTS
-----
.S"RAMPARAMASM" 1500 1520
SAVING RAMPARAMASM
.■

```

Wie funktioniert das Unterprogramm?

Im ersten Befehl wird wieder das X Register als Schleifenzähler mit dem Startwert 0 initialisiert.

Dann wird der erste Parameter (Zeichencode) aus der Speicherstelle \$1500 (dezimal 5376) gelesen und mit dem nächsten Befehl an die jeweilige Position im Videospeicher geschrieben.

Anschließend wird der Parameter für den Farbcode aus der Speicherstelle \$1501 (dezimal 5377) gelesen und an die jeweilige Position im Videospeicher geschrieben. Der Rest des Unterprogramms hat sich nicht verändert.

Wechseln Sie zurück nach BASIC, geben folgendes Programm ein und speichern es unter dem Namen „RAMPARAMBAS“ auf Diskette.

```
,1516 20 02 15 JSR 1502
,1519 60 RTS
-----
.S"RAMPARAMASM" 1500 1520
SAVING RAMPARAMASM
.X
READY.
10 POKE 5376,1
20 POKE 5377,7
30 SYS 5398
40 GET A$
50 IF A$="" THEN 40
60 POKE 5376,2
70 POKE 5377,0
80 SYS 5398
90 GET A$
100 IF A$="" THEN 90
110 POKE 5376,3
120 POKE 5377,3
130 SYS 5398
SAVE"RAMPARAMBAS",8
SAVING RAMPARAMBAS
READY.
```

Löschen Sie den Bildschirm, bewegen den Cursor einige Zeilen nach unten und starten das Programm mit RUN.

Das Ergebnis ist wiederum dasselbe, zuerst werden die gelben A's, dann die schwarzen B's und letztendlich die türkisen C's angezeigt.

Doch dieses mal haben wir kein reines Maschinenprogramm geschrieben, sondern ein BASIC-Programm, welches die Parameter für das Unterprogramm in den Speicherstellen \$1500 (dezimal 5376) und \$1501 (dezimal 5377) ablegt und dann das Unterprogramm mit SYS 5394 aufruft.

5.1.4 Übergabe der Parameter auf dem Stack

Diese Methode ist auf anderen Systemen (z.B. auf PC-Systemen) die Regel, doch hier, bei der Programmierung der CPU des C64, stellt sie eher die Ausnahme dar.

Sie kommt nur in Ausnahmefällen zur Anwendung, da sehr viel Code im Unterprogramm dazu verwendet werden muss, um Werte zwischen den Registern und dem Stack hin und her zu kopieren.

Hinzu kommt noch der bereits unter 5.1.3 angesprochene Nachteil was die Geschwindigkeit des Zugriffs betrifft.

Ich werde an dieser Stelle daher kein Programmbeispiel zu dieser Methode vorstellen, da dies im Moment keinen Mehrwert bringt. Stattdessen will ich die Methode dann vorstellen, wenn ihre Anwendung Sinn macht.

In der Praxis muss man von Fall zu Fall entscheiden, welche Art der Parameterübergabe die optimale ist und oft wird es vorkommen, dass man die Übergabeorte mischt.

Man könnte ja zwei Parameter in den Registern und weitere Parameter beispielsweise in der Zeropage zur Verfügung stellen.

Das ist das Schöne an der Maschinensprache, man hat weitreichende Entscheidungsfreiheiten über die einzelnen Abläufe.

Soweit so gut.

Aber etwas stört mich an den letzten beiden Beispielen noch. Diese Beispiele hatten wir auf ein BASIC-Programm und ein Maschinenprogramm aufgeteilt. Wenn wir das Programm starten wollen, müssen wir zuerst den SMON laden, dort das Assembler-Programm laden, dann zu Basic wechseln, dort das BASIC-Programm laden und dieses mit RUN starten.

Ziemlich umständlich, oder? Aber wie sieht die Lösung aus?

Na klar, wir verpacken das Maschinenprogramm in einen Basic-Loader und kombinieren diesen mit unserem BASIC-Programm. Doch das schaffen Sie mittlerweile sicher ohne meine Hilfe.

5.2 Beispiel-Programm

Zum krönenden Abschluss dieses Kapitels möchte ich Ihnen ein Beispielprogramm vorstellen, welches vieles, das wir bis jetzt gelernt haben, in sich vereint.

Was soll das Programm machen?

Es soll in der ersten Bildschirmzeile ein beliebiges Zeichen in einer beliebigen Farbe anzeigen. Dieses Zeichen soll man mit den beiden horizontalen Cursor-tasten nach rechts bzw. nach links bewegen können.

Falls das Zeichen am rechten oder linken Rand angekommen ist, soll keine Reaktion seitens des Programms erfolgen, wenn man es weiter in die jeweilige Richtung bewegen will.

Durch Betätigen der Taste „Q“ für Quit soll das Programm beendet werden können.

Welchen Zeichencode und Farbcode das Programm verwendet, legen wir als Parameter in den beiden Zeropage Adressen \$FB (dezimal 251) und \$FC (dezimal 252) ab.

In der Speicherstelle \$FD (dezimal 253) speichern wir laufend die Position mit, in der das Zeichen innerhalb der ersten Bildschirmzeile gerade angezeigt wird.

Hier das Assembler-Listing des Programms. Die Adressen am Beginn der Zeilen dienen nur dem Vergleich, ob Sie beim Eingeben mit der Adresse noch richtig unterwegs sind, diese daher bitte nicht eingeben.

Ich habe die Befehle hier so dargestellt, wie Sie einzugeben sind, und nicht wie sie vom SMON dann angezeigt werden.

Das Programm besteht aus drei Unterprogrammen, welche ich nun im Detail vorstellen werde.

Unterprogramm 1:

```
1500 LDX $FD
1502 CPY #$01
1504 BEQ $1513
1506 LDA #$20
1508 STA $0400,X
150B LDA #$06
150D STA $D800,X
1510 JMP $151D
1513 LDA $FB
1515 STA $0400,X
1518 LDA $FC
151A STA $D800,X
151D RTS
```

Dieses Unterprogramm dient dazu, das gewählte Zeichen innerhalb der obersten Bildschirmzeile in der gewünschten Farbe darzustellen. Aber nicht nur das, es bietet auch die Möglichkeit, das Zeichen an der aktuellen Position zu löschen. Denn wenn das Zeichen nach rechts oder nach links bewegt werden soll, muss es ja zuerst an der aktuellen Position gelöscht und dann an der neuen Position angezeigt werden.

Für die Unterscheidung, ob das Zeichen nun dargestellt oder gelöscht werden soll, habe ich das Y Register genutzt. Wenn es bei Eintritt in das Unterprogramm den Wert 1 enthält, dann soll das Zeichen in der gewünschten Farbe dargestellt werden. Enthält es jedoch einen anderen Wert, dann wird an die aktuelle Position ein Leerzeichen in blauer Farbe geschrieben, wodurch das Zeichen das sich bis dahin dort befunden hat, gelöscht wird.

Als Speicherstelle zum „Mitschreiben“ der aktuellen Zeichenposition habe ich die Speicherstelle \$FD (dezimal 253) gewählt, daher auch der erste Befehl LDX \$FD, welcher die aktuelle Position aus dieser Speicherstelle liest und in das X Register überträgt.

Als nächstes wird der Inhalt des Y Registers geprüft. Enthält es den Wert 1, geht es weiter bei der Programmadresse \$1513. Dort wird der Zeichencode aus der von uns zu diesem Zweck gewählten Speicherstelle \$FB (dezimal 251) gelesen und in den Videospeicher geschrieben.

Als Übergabeort für den Farbcode haben wir die Speicherstelle \$FC (dezimal 252) gewählt, daher wird diese durch den Befehl LDA \$FC ausgelesen und an die entsprechende Stelle im Videospeicher geschrieben. Danach geht's durch den Befehl RTS retour zum Aufrufer.

Enthält das Y Register jedoch nicht den Wert 1, dann wird der Code ab Programmadresse \$1506 ausgeführt, welcher das Leerzeichen in blauer Farbe ausgibt und dadurch das dort aktuell befindliche Zeichen löscht. Dann wird durch den Befehl JMP \$151D zum Befehl RTS gesprungen, wodurch das Ende des Unterprogrammes erreicht ist und zum Aufrufer zurückgesprungen wird.

Unterprogramm 2:

```
151E LDX $FD
1520 CPX #$27
1522 BEQ $1530
1524 LDY #$00
1526 JSR $1500
1529 INC $FD
152B LDY #$01
152D JSR $1500
1530 RTS
```

Dieses Unterprogramm ist dafür zuständig, das Zeichen nach rechts zu bewegen. Als erstes wird wieder die aktuelle Position aus der Speicherstelle \$FD gelesen und im nächsten Befehl wird geprüft, ob diese dem Wert \$27 (dezimal 39) entspricht. Dies ist die letzte Position in der Zeile und wir wollen, dass das Zeichen in rechter Richtung nur bis zu dieser Position bewegt werden kann.

Steht das Zeichen bereits an der letzten Position, wird gleich zum Befehl RTS an der Programmadresse \$1530 gesprungen, wodurch es ohne weitere Reaktion zurück zum Aufrufer geht.

Ansonsten starten wir mit der Bewegung des Zeichens nach rechts.

Dazu wird das Zeichen zunächst an seiner aktuellen Position entfernt. Das Y Register wird mit dem Wert 0 geladen und Unterprogramm 1 aufgerufen. Da das Y Register den Wert 0 enthält, wird das Zeichen an der aktuellen Position entfernt.

Dann wird die Position, welche in der Speicherstelle \$FD steht, durch den Befehl INC \$FD um 1 erhöht, denn das Zeichen soll ja um eine Stelle nach rechts bewegt werden.

Nun wird Unterprogramm 1 erneut aufgerufen, aber dieses mal schreiben wir vorher den Wert 1 in das Y Register, wodurch das Zeichen an der neuen Position angezeigt wird.

Unterprogramm 3:

```
1531 LDX $FD
1533 CPX #$00
1535 BEQ $1543
1537 LDY #$00
1539 JSR $1500
153C DEC $FD
153E LDY #$01
1540 JSR $1500
1543 RTS
```

Dieses Unterprogramm ist dafür zuständig, das Zeichen nach links zu bewegen. Als erstes wird wieder die aktuelle Position aus der Speicherstelle \$FD gelesen und im nächsten Befehl wird geprüft, ob diese dem Wert \$00 (dezimal 0) entspricht. Dies ist die erste Position in der Zeile und wir wollen, dass das Zeichen in linker Richtung nur bis zu dieser Position bewegt werden kann.

Steht das Zeichen bereits an der ersten Position, wird gleich zum Befehl RTS an der Programmadresse \$1543 gesprungen, wodurch es ohne weitere Reaktion seitens des Programms zurück zum Aufrufer geht.

Ansonsten starten wir mit der Bewegung des Zeichens nach links.

Dazu wird das Zeichen zunächst an seiner aktuellen Position entfernt. Das Y Register wird mit dem Wert 0 geladen und Unterprogramm 1 aufgerufen. Da das Y Register den Wert 0 enthält, wird das Zeichen an der aktuellen Position entfernt.

Dann wird die Position, welche in der Speicherstelle \$FD steht, durch den Befehl DEC \$FD um 1 vermindert, denn das Zeichen soll ja um eine Stelle nach links bewegt werden.

Nun wird Unterprogramm 1 erneut aufgerufen, aber dieses mal schreiben wir vorher den Wert 1 in das Y Register, wodurch das Zeichen an der neuen Position angezeigt wird.

Hauptprogramm:

1544 LDA #\$01

1546 STA \$FB

1548 LDA #\$07

154A STA \$FC

154C LDA #\$14

154E STA \$FD

1550 LDY #\$01

1552 JSR \$1500

1555 JSR \$FFE4

1558 BEQ \$1555

155A CMP #\$1D

155C BEQ \$1569

155E CMP #\$9D

1560 BEQ \$156F

1562 CMP #\$51

1564 BEQ \$1575

1566 JMP \$1555

1569 JSR \$151E

156C JMP \$1555

156F JSR \$1531

1572 JMP \$1555

1575 RTS

Speichern Sie das Programm mit dem Befehl

S“MOVECHAR“ 1500 1576

unter dem Namen „MOVECHAR“ auf Diskette ab, wechseln mit „X“ nach BASIC, löschen wie gehabt den Bildschirm, bewegen den Cursor um einige Zeilen nach unten und starten das Programm mit SYS 5444.



Das gelbe A wird nun an Startposition 20 angezeigt und Sie sollten es mit den horizontalen Cursortasten bis zum rechten und linken Bildschirmrand bewegen können.

Beenden können Sie das Programm durch Drücken der Taste Q.

Wie funktioniert das Hauptprogramm?

Der Startpunkt für das Programm liegt an der Programmadresse \$1544 (dezimal 5444)

Zuerst wird hier in diesem Beispiel der Zeichencode 1 (für den Buchstaben A) als Parameter in die Speicherstelle \$FB und der Farbcode 7 (für die Farbe Gelb) in die Speicherstelle \$FC geschrieben.

Als Startposition wählen wir beispielsweise die Mitte der Zeile, also die Position \$14 (dezimal 20), welche wir in die Speicherstelle \$FD schreiben.

Durch die Befehle an den Programmadressen \$1550 und \$1552 wird das Zeichen in der gewünschten Farbe auf die Startposition (in unserem Beispiel die Position 20) gesetzt. Zuerst wird das Y Register mit dem Wert 1 geladen und dann Unterprogramm 1 aufgerufen.

Durch die Befehle an den Programmadressen \$1555 und \$1558 wird die Tastatur, wie bereits beim vorherigen Programmbeispiel gezeigt, solange abgefragt bis der Benutzer eine Taste drückt.

Drückt der Benutzer die Cursortaste links, dann wird der Zeichencode dieser Taste (\$1D) in den Akkumulator geschrieben.

Durch den Befehl CMP #\$1D prüfen wir, ob der Akkumulator diesen Zeichencode enthält und wenn dies der Fall ist, geht's weiter bei Programmadresse \$1569.

Dort wird das Unterprogramm 2 aufgerufen, welches das Zeichen nach rechts bewegt.
Danach wird durch den Sprung an die Programmadresse \$1555 wieder die Tastatur abgefragt.

Drückt der Benutzer die Cursortaste rechts, dann wird der Zeichencode dieser Taste (\$9D) in den Akkumulator geschrieben.

Durch den Befehl CMP #\$9D prüfen wir, ob der Akkumulator diesen Zeichencode enthält und wenn dies der Fall ist, geht's weiter bei Programmadresse \$156F.

Dort wird das Unterprogramm 3 aufgerufen, welches das Zeichen nach links bewegt.
Danach wird durch den Sprung an die Programmadresse \$1555 wieder die Tastatur abgefragt.

Drückt der Benutzer die Taste Q, dann wird der Zeichencode dieser Taste (\$51) in den Akkumulator geschrieben.

Durch den Befehl CMP #\$_51 prüfen wir, ob der Akkumulator diesen Zeichencode enthält und wenn dies der Fall ist, geht's weiter bei Programmadresse \$1575.

Dort steht schließlich der Befehl RTS, wodurch das Programm beendet wird.

Einen Aspekt des Programms müssen wir noch besprechen.

Wenn wir das Programm an der Adresse \$1544 starten, dann wird durch den Code von \$1544 bis \$154E festgelegt, dass als Zeichen der Buchstabe A in gelber Farbe verwendet werden soll.

Zusätzlich wird festgelegt, dass die Ausgangsposition bei 20 liegt.

Die ursprüngliche Intention war aber ja eigentlich, dass wir eben genau diese Einstellungen durch Parameter steuern können. Durch diese Festlegung auf das gelbe A an Position 20 umgehen wir hier ja diese Einstellungs-Möglichkeiten.

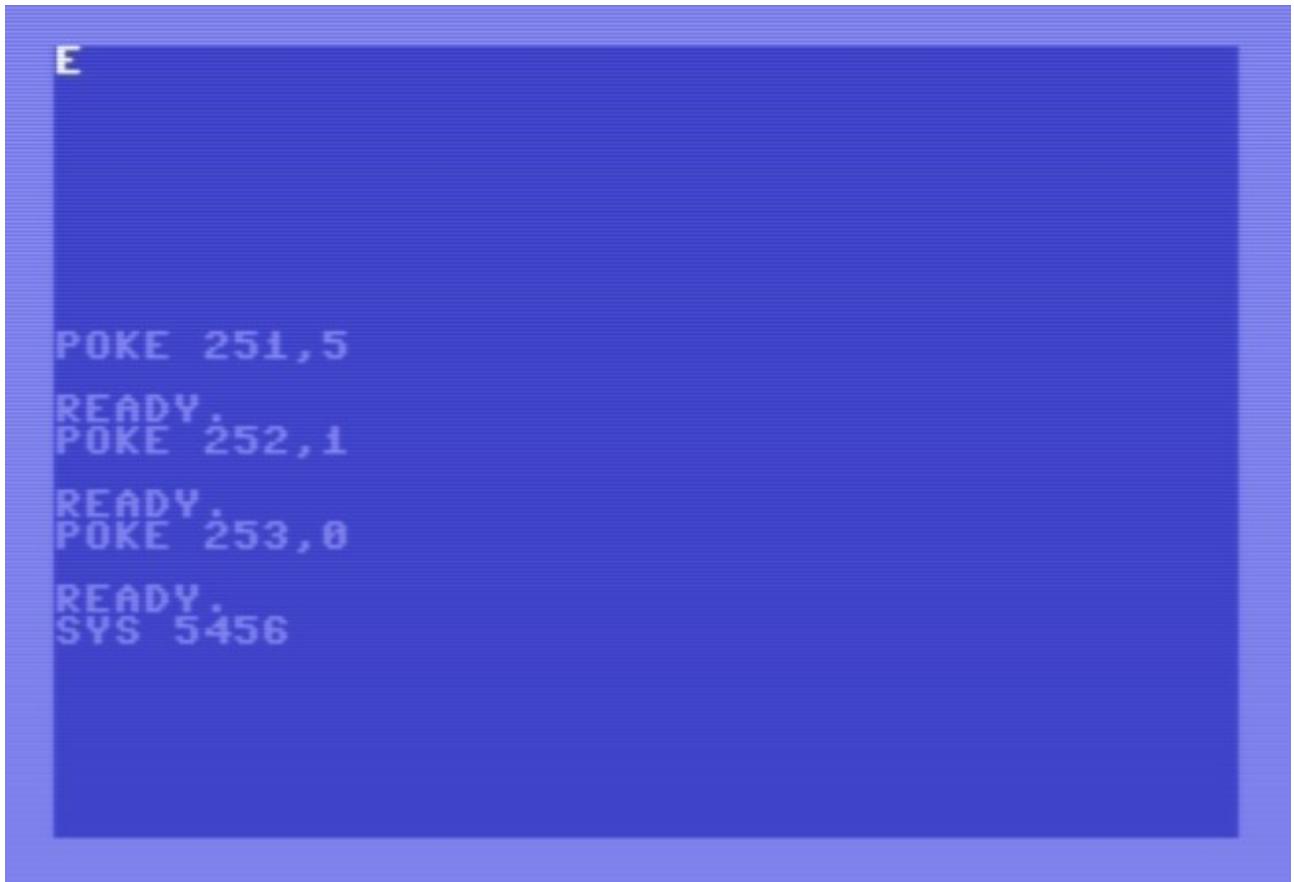
Das ist richtig, aber wir haben die Möglichkeit, das Programm nicht an der Adresse \$1544 zu starten, sondern erst an der Adresse \$1550 (dezimal 5456).

Dadurch wird der Teil, welche die obigen Einstellungen vornimmt, nicht ausgeführt und somit auch keine Werte in Speicherstellen \$FB, \$FC und \$FD geschrieben.

Nun müssen wir jedoch selbst dafür sorgen, dass in den Speicherstellen \$FB (Zeichencode), \$FC (Farbcode) und \$FD (Position) unsere gewünschten Werte stehen.

Dies können wir leicht über POKE-Befehle in diese Speicherstellen erreichen.

Nach dem Start des Programms durch SYS 5456 sollten sie folgendes Ergebnis erhalten und das weiße E horizontal bewegen können.



Hier wird der Zeichencode 5 (für den Buchstaben E) in die Speicherstelle 251 (\$FB), der Farbcode 1 (für die Farbe Weiß) in die Speicherstelle 252 (\$FC) und die 0 für die gewünschte Startposition in die Speicherstelle 253 (\$FD) geschrieben.

Und genau aus diesen Speicherstellen lesen sich die Unterprogramme die entsprechenden Parameter aus.

Führen Sie diese Einstellungen nicht durch, dann werden die Werte verwendet, welche sich aktuell in diesen Speicherstellen befinden und das optische Ergebnis ist nicht vorhersehbar.

Wenn man das Programm wie vorhin mit SYS 5444 startet, ist es nicht nötig, Parameter einzustellen, sondern das Programm nimmt Standard-Einstellungen in Form des gelben A's an Startposition 20 vor.

So wie immer kann man durch die abschließende Erstellung eines Basic-Loaders das Laden und Starten des Programms einfacher gestalten, da man nicht immer den Umweg über den SMON nehmen muss.

Abschließend habe ich noch ein paar interessante Informationen zum Befehl SYS in Zusammenhang mit den CPU-Registern für Sie.

Auf Seite 12 haben wir nach Beendigung eines Maschinenprogramms, welches mit SYS aufgerufen wurde, die Inhalte des Akkumulators, des X Registers und des Y Registers über die Speicherstellen 780, 781 und 782 ausgelesen.

Dies funktioniert auch in die Gegenrichtung, d.h. man kann vor dem Aufruf eines Maschinenprogramms über den Befehl SYS, durch POKEs in diese Speicherstellen angeben, welche Werte beim Aufruf von SYS in diese Register geladen werden sollen.

Der Befehl SYS holt sich dann die Werte aus diesen Speicherstellen und lädt die Register mit diesen Werten, bevor mit der Ausführung des Maschinenprogramms begonnen wird.

Das heißt, man könnte diese Vorgehensweise auch zur Parameterübergabe an ein Maschinenprogramm nutzen und es wäre eine alternative Herangehensweise an die Parameterübergabe in den CPU-Registern, welche bereits beschrieben wurde.

Hier nochmal als Zusammenfassung die Zuordnung der Register zu den Speicherstellen:

| Register | Speicherstelle |
|----------------|----------------|
| Akkumulator | 780 |
| X Register | 781 |
| Y Register | 782 |
| Statusregister | 783 |

Spielen wir die Sache mal durch.

Geben Sie folgendes Programm ab Adresse \$1500 ein:

```
.D 1500
;1500 85 FB      STA    FB
;1502 86 FC      STX    FC
;1504 84 FD      STY    FD
;1506 A9 64      LDA    #64
;1508 A2 C8      LDX    #C8
;150A A0 FF      LDY    #FF
;150C 60        RTS

.■
```

und speichern es mit dem Befehl S“REGTESTASM“ 1500 150D

auf Diskette.

Dann wechseln Sie zurück zu BASIC und geben folgendes Programm ein:

```
10 POKÉ 780,10
20 POKÉ 781,20
30 POKÉ 782,30
40 SYS 5376
50 PRINT "PEEK (251) = ";PEEK(251)
60 PRINT "PEEK (252) = ";PEEK(252)
70 PRINT "PEEK (253) = ";PEEK(253)
80 PRINT "PEEK (780) = ";PEEK(780)
90 PRINT "PEEK (781) = ";PEEK(781)
100 PRINT "PEEK (782) = ";PEEK(782)
READY.
```

Speichern Sie es unter dem Namen „REGTEST“ ab.

Nach dem Start mit RUN sollten Sie folgendes Ergebnis erhalten:

```
RUN
PEEK (251) = 10
PEEK (252) = 20
PEEK (253) = 30
PEEK (780) = 100
PEEK (781) = 200
PEEK (782) = 255
```

```
READY.
```

Erklärung zum Ablauf des Programms:

In den Zeilen 10-30 werden die Speicherstellen 780, 781 und 782 mit den Werten 10, 20 und 30 befüllt.

Dann wird mittels des Befehls SYS das Maschinenprogramm an der Adresse \$1500 aufgerufen. Doch bevor mit der Ausführung begonnen wird, hat der Befehl SYS die Inhalte der drei genannten Speicherstellen in den Akkumulator, das X Register und das Y Register kopiert.

Das Maschinenprogramm kopiert die Registerinhalte „zum Beweis“ in die Speicherstellen \$FB (dezimal 251), \$FC (dezimal 252) und \$FD (dezimal 253).

Dann lädt es die drei Register mit den beliebig gewählten Werten \$64 (dezimal 100), \$C8 (dezimal 200) und \$FF (dezimal 255)

Abschließend kehrt das Programm durch den Befehl RTS zum Aufrufer zurück. Doch zuvor werden die Inhalte der drei Register in die entsprechenden Speicherstellen 780, 781 und 782 kopiert.

Wenn Sie das BASIC-Programm starten, werden die Inhalte der betroffenen Speicherstellen ausgegeben.

Die ersten drei Ausgaben beziehen sich auf die Speicherstellen \$FB (dezimal 251), \$FC (dezimal 252) und \$FD (dezimal 253). Sie enthalten die Werte 10, 20 und 30, wodurch bewiesen ist, dass der Befehl SYS vor der Ausführung des Maschinenprogramms tatsächlich die Inhalte der Speicherstellen 780, 781 und 782 in die Register übernommen hat.

Die nächsten drei Ausgaben beweisen, dass vor der Rückkehr aus dem Unterprogramm die Registerinhalte \$64 (dezimal 100), \$C8 (dezimal 200) und \$FF (dezimal 255) in die Speicherstellen 780, 781 und 782 übernommen wurden.

So, ich hoffe, dass Sie durch die Arbeit mit Unterprogrammen nun noch neugieriger geworden sind und auch weiterhin Interesse daran haben, das Programmieren in Assembler zu erlernen.

6 Zahlsysteme

Auf der Welt gibt es unzählige Sprachen, von denen jede ihren eigenen Wortschatz, ihre eigene Grammatik und ihre eigenen Symbole, zur bildlichen Darstellung dessen, was man ausdrücken will, mitbringt.

Die deutsche Sprache verwendet neben einigen Sonderzeichen (z.B. Ä, Ü, Ö) beispielsweise die Symbole A bis Z bzw. a bis z, um daraus Wörter zusammenzusetzen. Chinesisch verwendet wiederum völlig andere Symbole, ebenso wie Sprachen aus dem arabischen Raum.

Sprachen verwenden wir, um Worte auszudrücken, z.B. "Hallo", "Computer" oder "Country"

Zahlensysteme werden im Gegensatz dazu verwendet, um Zahlenwerte auszudrücken.

Auch hier gibt es "Sprachen", die jeweils ihre eigenen Symbole (Ziffern) mitbringen, um einen Zahlenwert auszudrücken.

6.1 Dezimalsystem, Binärsystem, Hexadezimalsystem

Das Dezimalsystem beispielsweise, welches wir Menschen hauptsächlich verwenden, bringt 10 Ziffern mit, um einen Zahlenwert auszudrücken. Diese lauten 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9

Außerdem bringt es noch das Symbol "-" zur Darstellung von negativen Zahlen mit.

Das Binärsystem wiederum bringt nur zwei Ziffern mit, nämlich 0 und 1.

Im Hexadezimalsystem gibt es sogar 16 Ziffern, nämlich neben den Ziffern 0 bis 9 noch die Buchstaben A-F.

Es gibt auch noch andere Zahlensysteme, z.B. das Oktalsystem welches die Ziffern 0 bis 7 mitbringt.

Diese Ziffern ergeben zusammengesetzt einen bestimmten Zahlenwert, so wie man Buchstaben zu einem Wort zusammensetzt.

Und ebenso wie ein bestimmtes Wort in unterschiedlichen Sprachen unterschiedlich dargestellt wird, wird ein Zahlenwert von einem Zahlensystem zum anderen ebenfalls unterschiedlich dargestellt.

Das deutsche Wort "wo" beispielsweise lautet in englischer Sprache "where".

Übertragen auf Zahlensysteme würde ein Zahlenwert, der im dezimalen Zahlensystem als 200 dargestellt wird, im binären Zahlensystem als %11001000 und im hexadezimalen Zahlensystem als \$C8 dargestellt werden.

Hinweis:

In den folgenden Ausführungen werde ich klarerweise sehr oft Darstellungen von Werten in unterschiedlichen Zahlensystemen verwenden.

Damit ich die Schreibweisen nicht immer extra erwähnen muss, gebe ich ab jetzt Binärzahlen mit führendem "%", Hexadezimalzahlen mit führendem "\$" und Dezimalzahlen ohne führendes Symbol an.

Keine Sorge, wir kommen auf die "Übersetzungen" zwischen den einzelnen Zahlensystemen noch genau zu sprechen, der soeben genannte binäre Zahlenwert %11001000 und hexadezimale Zahlenwert \$C8 sollten nur mal als Beispiel für unterschiedliche Darstellungen in unterschiedlichen Zahlensystemen dienen.

Nichtsdestotrotz muss ich ein wenig vorgreifen und die Übersetzungen der Ziffern zwischen dem Dezimalsystem, dem Binärsystem und dem Hexadezimalsystem gegenüberstellen.

Nehmen Sie diese Übersetzungen für den Moment bitte mal so hin. Wie man eine Zahl von einem Zahlensystem ins andere übersetzt, werde ich noch im Detail erklären.

| dezimal | hexadezimal | binär |
|---------|-------------|-------|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 10 |
| 3 | 3 | 11 |
| 4 | 4 | 100 |
| 5 | 5 | 101 |
| 6 | 6 | 110 |
| 7 | 7 | 111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

Die farbig markierten Bereiche stellen die Ziffern in dem jeweiligen Zahlensystem dar.

Wie bereits erwähnt, bestehen Zahlenwerte, unabhängig vom Zahlensystem, aus einer oder mehreren Ziffern. Die Positionen an denen sich die Ziffern innerhalb des Zahlenwertes befinden, möchte ich ab jetzt als Stellen bezeichnen, welche ich für die nachfolgenden Ausführungen von rechts und mit 0 beginnend aufsteigend durchnummieren möchte.

Beispiel 1:

Beginnen wir mit unserem vertrauten Dezimalsystem und nehmen als Beispiel die Zahl 5376.

Diese Zahl besteht aus 4 Ziffern.

An Stelle Nr. 0 haben wir hier die Ziffer 6, an Stelle Nr. 1 die Ziffer 7, an Stelle Nr. 2 die Ziffer 3 und an Stelle Nr. 3 die Ziffer 5.

Beispiel 2:

Die binäre Zahl %10110

Diese Zahl besteht aus 5 Ziffern welche von 0 bis 4 nummeriert sind.

An Stelle Nr. 0 haben wir hier die Ziffer %0, an Stelle Nr. 1 die Ziffer %1, an Stelle Nr. 2 die Ziffer %1, an Stelle Nr. 3 die Ziffer %0 und an Stelle Nr. 4 die Ziffer %1.

Beispiel 3:

Die hexadezimale Zahl \$C8

An Stelle Nr. 0 haben wir die Ziffer \$8 und an Stelle Nr. 1 die Ziffer \$C.

Doch was unterscheidet nun die Zahlensysteme voneinander wenn man mal von der unterschiedlichen Anzahl und Art der Ziffern absieht?

Der Unterschied liegt in der Wertigkeit der einzelnen Stellen. Jedes Zahlensystem hat einen bestimmten Basiswert, auf dem die Wertigkeit der einzelnen Stellen basieren.

Im Dezimalsystem ist dies die Zahl 10, im Binärsystem die Zahl 2 und im hexadezimalen System die Zahl 16. Im angesprochenen Oktalsystem wäre das die Zahl 8, man kann also mit jedem Basiswert ein neues Zahlensystem bilden, z.B. auch mit dem Basiswert 3 wenn man will.

Die Wertigkeiten der einzelnen Stellen ergeben sich aus der Potenz:

Basiszahl des Zahlensystems hoch Nummer der Stelle

Wertigkeiten im Dezimalsystem:

10 hoch 0 = 1
10 hoch 1 = 10
10 hoch 2 = 100
10 hoch 3 = 1000
10 hoch 4 = 10000
...

Wertigkeiten im Binärsystem:

2 hoch 0 = 1
2 hoch 1 = 2
2 hoch 2 = 4
2 hoch 3 = 8
2 hoch 4 = 16
...

Wertigkeiten im Hexadezimalsystem:

16 hoch 0 = 1
16 hoch 1 = 16
16 hoch 2 = 256
16 hoch 3 = 4096
16 hoch 4 = 65536
...

OK, was haben wir bis jetzt? Wir haben eine Reihe aus Ziffern vor uns, wir wissen aus der obigen Tabelle welchen dezimalen Wert jede Ziffer hat.

Im Falle einer Binärzahl entweder den dezimalen Wert 0 oder 1 und im Falle einer Hexadezimalzahl ein dezimalen Wert zwischen 0 und 15. Die hexadezimale Ziffer \$D würde laut der Tabelle dem dezimalen Wert 13 entsprechen.

Wir wissen außerdem, welche dezimale Wertigkeit eine bestimmte Stelle im jeweiligen Zahlensystem hat (siehe oben)

6.2 Umrechnen zwischen Zahlensystemen

Wie kommen wir nun zum dezimalen Gesamtwert einer Zahl die uns als Zahl aus einem anderen Zahlensystem vorliegt?

Um den dezimalen Wert einer Zahl zu erhalten, geht man die Zahl Stelle für Stelle durch, multipliziert die dezimale Wertigkeit an dieser Stelle mit der dezimalen Ziffer an dieser Stelle und am Ende summiert man diese Produkte auf, um den dezimalen Wert zu erhalten, den die Zahl darstellt.

Spielen wir die Sache mal anhand der oben genannten dezimalen Zahl 5376 durch. Klar, wir wissen natürlich was rauskommt, nämlich 5376, aber nehmen wir mal an, wir sehen nur die dezimalen Ziffern 5, 3, 7 und 6 vor uns und wollen den Gesamtwert ausrechnen.

Wir gehen vor wie vorhin beschrieben, d.h. wir bilden jeweils das Produkt der Ziffer und der Wertigkeit an dieser Stelle und am Ende summieren wir diese Produkte zum Gesamtwert auf.

$$\begin{aligned}6 * (10 \text{ hoch } 0) &= 6 * 1 = 6 \\7 * (10 \text{ hoch } 1) &= 7 * 10 = 70 \\3 * (10 \text{ hoch } 2) &= 3 * 100 = 300 \\5 * (10 \text{ hoch } 3) &= 5 * 1000 = 5000\end{aligned}$$

Nun summieren wir diese Einzelwerte:

$$6 + 70 + 300 + 5000 = 5376$$

Nun wenden wir dasselbe Schema auf die oben genannte Binärzahl %10110 an.

$$\begin{aligned}
 \%0 * (2 \text{ hoch } 0) &= 0 * 1 = 0 \\
 \%1 * (2 \text{ hoch } 1) &= 1 * 2 = 2 \\
 \%1 * (2 \text{ hoch } 2) &= 1 * 4 = 4 \\
 \%0 * (2 \text{ hoch } 3) &= 0 * 8 = 0 \\
 \%1 * (2 \text{ hoch } 4) &= 1 * 16 = 16
 \end{aligned}$$

Nun summieren wir wieder die Einzelwerte:

$$0 + 2 + 4 + 0 + 16 = 22$$

Wenden wir das Schema nun auch auf die hexadezimale Zahl \$C8 an.

$$\begin{aligned}
 \$8 * (16 \text{ hoch } 0) &= 8 * 1 = 8 \\
 \$C * (16 \text{ hoch } 1) &= 12 * 16 = 192
 \end{aligned}$$

Summe der Einzelwerte:

$$8 + 192 = 200$$

Jetzt haben wir gelernt wie wir eine Binärzahl oder eine Hexadezimalzahl ins Dezimalsystem umrechnen können.

Spielen wir das zur Übung nochmal anhand zweier Beispiele durch:

Beispiel 1:

Binärzahl %1001111000101

$$\begin{aligned}
 \%1 * (2 \text{ hoch } 0) &= 1 * 1 = 1 \\
 \%0 * (2 \text{ hoch } 1) &= 0 * 2 = 0 \\
 \%1 * (2 \text{ hoch } 2) &= 1 * 4 = 4 \\
 \%0 * (2 \text{ hoch } 3) &= 0 * 8 = 0 \\
 \%0 * (2 \text{ hoch } 4) &= 0 * 16 = 0 \\
 \%0 * (2 \text{ hoch } 5) &= 0 * 32 = 0 \\
 \%1 * (2 \text{ hoch } 6) &= 1 * 64 = 64 \\
 \%1 * (2 \text{ hoch } 7) &= 1 * 128 = 128 \\
 \%1 * (2 \text{ hoch } 8) &= 1 * 256 = 256 \\
 \%1 * (2 \text{ hoch } 9) &= 1 * 512 = 512 \\
 \%0 * (2 \text{ hoch } 10) &= 0 * 1024 = 0 \\
 \%0 * (2 \text{ hoch } 11) &= 0 * 2048 = 0 \\
 \%1 * (2 \text{ hoch } 12) &= 1 * 4096 = 4096
 \end{aligned}$$

$$1 + 0 + 4 + 0 + 0 + 0 + 64 + 128 + 256 + 512 + 0 + 0 + 4096 = 5061$$

Beispiel 2:

Hexadezimalzahl \$FA37

$$\begin{aligned} \$7 * (16 \text{ hoch } 0) &= 7 * 1 = 7 \\ \$3 * (16 \text{ hoch } 1) &= 3 * 16 = 48 \\ \$A * (16 \text{ hoch } 2) &= 10 * 256 = 2560 \\ \$F * (16 \text{ hoch } 3) &= 15 * 4096 = 61440 \\ 7 + 48 + 2560 + 61440 &= 64055 \end{aligned}$$

Doch wie sieht der umgekehrte Weg aus? Wie kann man eine Dezimalzahl in eine Binärzahl oder eine Hexadezimalzahl umrechnen?

Kurzbeschreibung die für jedes gewünschte Ziel-Zahlensystem gilt:

Man dividiert die Dezimalzahl laufend durch die Basis des gewünschten Ziel-Zahlensystems und schreibt den Rest, der bei der Division bleibt auf (in der Darstellung des Ziel-Zahlensystems).

Das Ergebnis der Division nimmt man als Ausgangsbasis für die nächste Division. Man dividiert so lange, bis das Ergebnis der Division gleich 0 ist. Dann schreibt man die Restwerte von unten beginnend nebeneinander und hat das gewünschte Ergebnis.

Beispiel 1:

Umrechnen der Dezimalzahl 2348 in das Binärsystem.

| | |
|-----------------|---------|
| 2348 : 2 = 1174 | Rest %0 |
| 1174 : 2 = 587 | Rest %0 |
| 587 : 2 = 293 | Rest %1 |
| 293 : 2 = 146 | Rest %1 |
| 146 : 2 = 73 | Rest %0 |
| 73 : 2 = 36 | Rest %1 |
| 36 : 2 = 18 | Rest %0 |
| 18 : 2 = 9 | Rest %0 |
| 9 : 2 = 4 | Rest %1 |
| 4 : 2 = 2 | Rest %0 |
| 2 : 2 = 1 | Rest %0 |
| 1 : 2 = 0 | Rest %1 |

Nun schreiben wir von unten beginnen die Restwerte nebeneinander und kommen auf das Ergebnis:

%100100101100

Beispiel 2:

Umrechnen der Dezimalzahl 55327 in das Hexadezimalsystem.

| | |
|-------------------|-----------------------|
| 55327 : 16 = 3457 | Rest \$F (dezimal 15) |
| 3457 : 16 = 216 | Rest \$1 (dezimal 1) |
| 216 : 16 = 13 | Rest \$8 (dezimal 8) |

$13 : 16 = 0$

Rest \$D (dezimal 13)

Nun schreiben wir wie vorhin die Restwerte von unten beginnend nebeneinander und kommen auf das Ergebnis:

\$D81F

Beispiel 3:

Umrechnen der Dezimalzahl 12 in das Binärsystem.

| | |
|--------------|---------|
| $12 : 2 = 6$ | Rest %0 |
| $6 : 2 = 3$ | Rest %0 |
| $3 : 2 = 1$ | Rest %1 |
| $1 : 2 = 0$ | Rest %1 |

Das Ergebnis ist %1100 und wenn Sie einen kurzen Blick in die Tabelle werfen, in der die Ziffern der Zahlensysteme gegenübergestellt wurden, werden Sie sehen, dass dies korrekt ist:

| | | |
|----|---|------|
| 12 | C | 1100 |
| 11 | D | 1101 |

Beispiel 4:

Umrechnen der Dezimalzahl 1024 in das Hexadezimalsystem.

| | |
|------------------|----------------------|
| $1024 : 16 = 64$ | Rest \$0 (dezimal 0) |
| $64 : 16 = 4$ | Rest \$0 (dezimal 0) |
| $4 : 16 = 0$ | Rest \$4 (dezimal 4) |

Nun wieder die Restwerte von unten beginnend nebeneinander schreiben und wir kommen auf das Ergebnis:

\$400

Eine Hexadezimalzahl in eine Binärzahl umzuwandeln und umgekehrt ist recht einfach wie Sie gleich sehen werden.

Das Hexadezimalsystem besitzt wie wir nun wissen 16 Ziffern, 0-9 und A-F.

Wieviele Werte kann man mit einer Ziffer im Hexadezimalsystem darstellen? Richtig, 16.

Wieviele Ziffern braucht man dafür im Binärsystem? Richtig, 4, denn \$F entspricht %1111
Um den höchsten Wert, welchen man mit einer hexadezimalen Ziffer darstellen kann (\$F), binär darzustellen, braucht es also eine vierstellige Binärzahl.

Um also eine Hexadezimalzahl in eine Binärzahl umzuwandeln, braucht man nur die binären Gegenstücke ihrer Ziffern aus der Tabelle nebeneinander aufschreiben.

Dabei gilt es jedoch folgendes zu beachten:

Sollte das binäre Gegenstück der jeweiligen hexadezimalen Ziffer weniger als 4 Stellen haben, dann muss diese mit 0-Stellen auf diese Länge von 4 Stellen von links aufgefüllt werden, damit die Reihenfolge der entsprechenden Stellen wieder zusammenpasst.

Hier ein Beispiel:

Wir wollen die Hexadezimalzahl \$AFDE ins Binärsystem umwandeln.

\$A = binär %1010

\$F = binär %1111

\$D = binär %1101

\$E = binär %1110

Nun schreiben wir binären Gegenstücke der Ziffern nebeneinander.

| \$A | \$F | \$D | \$E |
|------|------|------|------|
| 1010 | 1111 | 1101 | 1110 |

Das binäre Gegenstück zu \$AFDE lautet also %101011111011110

Nun ein Beispiel bei dem die binären Gegenstücke der hexadezimalen Ziffern teilweise weniger als 4 Stellen haben.

Wir wollen die Hexadezimalzahl \$3D7F ins Binärsystem umwandeln.

\$3 = %11

\$D = \$1101

\$7 = %111

\$F = %1111

| \$3 | \$D | \$7 | \$F |
|-----|------|-----|------|
| 11 | 1101 | 111 | 1111 |

Das ergibt nebeneinander geschrieben 111101111111

Rechnen wir diese Binärzahl ins Dezimalsystem um:

$$\%1 * (2 \text{ hoch } 0) = 1 * 1 = 1$$

$$\%1 * (2 \text{ hoch } 1) = 1 * 2 = 2$$

$$\%1 * (2 \text{ hoch } 2) = 1 * 4 = 4$$

$$\%1 * (2 \text{ hoch } 3) = 1 * 8 = 8$$

$$\%1 * (2 \text{ hoch } 4) = 1 * 16 = 16$$

$$\%1 * (2 \text{ hoch } 5) = 1 * 32 = 32$$

$$\%1 * (2 \text{ hoch } 6) = 1 * 64 = 64$$

$$\%1 * (2 \text{ hoch } 7) = 1 * 128 = 128$$

$$\%0 * (2 \text{ hoch } 8) = 0 * 256 = 0$$

$$\%1 * (2 \text{ hoch } 9) = 1 * 512 = 512$$

$$\%1 * (2 \text{ hoch } 10) = 1 * 1024 = 1024$$

$$\begin{aligned}\%1 * (2 \text{ hoch } 11) &= 1 * 2048 = 2048 \\ \%1 * (2 \text{ hoch } 12) &= 1 * 4096 = 4096\end{aligned}$$

$$1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 + 0 + 512 + 1024 + 2048 + 4096 = 7935 \text{ oder } \$1EFF$$

Dies entspricht jedoch nicht unserer ursprünglichen Zahl \$3D7F

Daher müssen wir folgende Korrektur vornehmen:

| \$3 | \$D | \$7 | \$F |
|------|------|------|------|
| 0011 | 1101 | 0111 | 1111 |

$$\begin{aligned}\%1 * (2 \text{ hoch } 0) &= 1 * 1 = 1 \\ \%1 * (2 \text{ hoch } 1) &= 1 * 2 = 2 \\ \%1 * (2 \text{ hoch } 2) &= 1 * 4 = 4 \\ \%1 * (2 \text{ hoch } 3) &= 1 * 8 = 8 \\ \%1 * (2 \text{ hoch } 4) &= 1 * 16 = 16 \\ \%1 * (2 \text{ hoch } 5) &= 1 * 32 = 32 \\ \%1 * (2 \text{ hoch } 6) &= 1 * 64 = 64 \\ \%0 * (2 \text{ hoch } 7) &= 0 * 128 = 0 \\ \%1 * (2 \text{ hoch } 8) &= 1 * 256 = 256 \\ \%0 * (2 \text{ hoch } 9) &= 0 * 512 = 0 \\ \%1 * (2 \text{ hoch } 10) &= 1 * 1024 = 1024 \\ \%1 * (2 \text{ hoch } 11) &= 1 * 2048 = 2048 \\ \%1 * (2 \text{ hoch } 12) &= 1 * 4096 = 4096 \\ \%1 * (2 \text{ hoch } 13) &= 1 * 8192 = 8192 \\ \%0 * (2 \text{ hoch } 14) &= 0 * 16384 = 0 \\ \%0 * (2 \text{ hoch } 15) &= 0 * 32768 = 0\end{aligned}$$

$$1 + 2 + 4 + 8 + 16 + 32 + 64 + 0 + 256 + 0 + 1024 + 2048 + 4096 + 8192 + 0 + 0 = 15743 \text{ bzw. nun korrekt } \$3D7F.$$

Wie sieht es umgekehrt aus, also vom Binärsystem ins Hexadezimalsystem?

Angenommen wir wollen die Binärzahl %10011100001010 ins Hexadezimalsystem umwandeln.

Dann teilen wir die Binärzahl von rechts beginnend in Gruppen von 4 Ziffern auf.

10 0111 0000 1010

Am linken Ende reichen die Stellen nicht aus um eine Vierergruppe zu bilden, daher füllen wir sie mit Nullen auf.

0010 0111 0000 1010

Nun gehen wir den umgekehrten Weg wie vorhin bei der Umrechnung einer Hexadezimalzahl ins Binärsystem:

| | | | |
|------|------|------|------|
| 0010 | 0111 | 0000 | 1010 |
| \$2 | \$7 | \$0 | \$A |

Das Ergebnis lautet: \$270A oder dezimal 9994.

Rechnen wir nach:

$$\begin{aligned}
 \$A * (16 \text{ hoch } 0) &= 10 * 1 = 10 \\
 \$0 * (16 \text{ hoch } 1) &= 0 * 16 = 0 \\
 \$7 * (16 \text{ hoch } 2) &= 7 * 256 = 1792 \\
 \$2 * (16 \text{ hoch } 3) &= 2 * 4096 = 8192
 \end{aligned}$$

$$10 + 0 + 1792 + 8192 = 9994$$

Und zur Übung und Kontrolle rechnen wir das gleich ins Binärsystem um:

| | |
|-----------------|---------|
| 9994 : 2 = 4997 | Rest %0 |
| 4997 : 2 = 2498 | Rest %1 |
| 2498 : 2 = 1249 | Rest %0 |
| 1249 : 2 = 624 | Rest %1 |
| 624 : 2 = 312 | Rest %0 |
| 312 : 2 = 156 | Rest %0 |
| 156 : 2 = 78 | Rest %0 |
| 78 : 2 = 39 | Rest %0 |
| 39 : 2 = 19 | Rest %1 |
| 19 : 2 = 9 | Rest %1 |
| 9 : 2 = 4 | Rest %1 |
| 4 : 2 = 2 | Rest %0 |
| 2 : 2 = 1 | Rest %0 |
| 1 : 2 = 0 | Rest %1 |

Wodurch wir dann wieder zu der Binärzahl %10011100001010 kommen.

Bevor wir nun zur Addition von Binärzahlen kommen, habe ich eine sehr gute Nachricht für Sie, denn die folgenden Programmbeispiele werden die letzten sein, zu deren Ausführung wir den SMON verwenden werden.

Der SMON ist zweifelsfrei ein sehr gutes Programm, aber für die Entwicklung größerer Assembler-Programme ist er denkbar ungeeignet, weil seine Stärken eindeutig woanders liegen.

Halten Sie also noch ein wenig durch!

Ab dem nächsten Kapitel, in dem es um das Thema Sprites gehen wird, werden wir dann den wesentlich komfortableren und in der Praxis sehr beliebten Turbo Macro Pro verwenden und Sie werden sehen, dass das Programmieren in Assembler dann gleich noch viel mehr Spaß macht, da der Turbo Macro Pro beispielsweise über einen komfortablen Editor für die Erstellung unserer Programme verfügt.

Wir werden uns dann auch nicht mehr mit Speicheradressen in Form von Zahlenwerten herumschlagen müssen, sondern können stattdessen aussagekräftige Namen verwenden, sodass die Lesbarkeit der Programme um ein Vielfaches besser wird.

Soviel schon mal als kleiner Vorgeschmack und Motivation zum Durchhalten, doch nun weiter im Text.

6.3 Addition von Binärzahlen

Im Grunde funktioniert die Addition im Binärsystem genau gleich wie wir es vom Dezimalsystem her kennen.

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$0 + 1 = 1$$

$$1 + 1 = 0 \text{ und Übertrag } 1 \text{ auf die nächste Stelle}$$

Falls wir folgende Situation haben:

$$1 + 1 + \text{Übertrag } 1, \text{ dann ergibt das } 1 \text{ und Übertrag } 1 \text{ auf die nächste Stelle}$$

Beispiel 1

Addieren wir mal die beiden Binärzahlen %00111010 (dezimal 58) und %00010011 (dezimal 19). Das ergibt %01001101 (dezimal 77).

Folgende Tabelle stellt diese Addition dar. In der ersten Zeile stehen die Ziffern der ersten binären Zahl, in der zweiten Zeile stehen die Ziffern der zweiten binären Zahl und das Ergebnis der Addition ist gelb markiert in der vierten Zeile dargestellt.

In der dritten Zeile wird durch die hellblauen Zellen hervorgehoben, dass hier ein Übertrag stattgefunden hat.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| | 1 | 1 | | | 1 | | |
| 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |

Beginnen wir von rechts mit der Addition.

$$0 + 1 = 1$$

$$1 + 1 = 0 + \text{Übertrag } 1$$

$$0 + 0 + \text{Übertrag } 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0 + \text{Übertrag } 1$$

$$1 + 0 + \text{Übertrag } 1 = 0 + \text{Übertrag } 1$$

$$0 + 0 + \text{Übertrag } 1 = 1$$

$$0 + 0 = 0$$

So sieht der zugehörige Assembler-Code aus, als Ergebnis ergibt sich im Akkumulator der Wert \$4D (dezimal 77 bzw. wie oben in der Tabelle gelb markiert binär %01001101)

Was es mit dem Befehl CLC auf sich hat, erkläre ich noch. Den Befehl ADC kennen sie bereits, er addiert einen Wert zum Inhalt des Akkumulators.

```

.A 1500
1500 18      CLC
1501 A9 3A    LDA #3A
1503 69 13    ADC #13
1505 00      BRK
1506 F
;1500 18      CLC
;1501 A9 3A    LDA #3A
;1503 69 13    ADC #13
;1505 00      BRK
-----
.G 1500
PC  SR  AC  XR  YR  SP  NU-BDIZC
;1506 30 4D 00 00 F6  00110000
.■

```

Beispiel 2

Addieren wir nun die beiden Binärzahlen %01110110 (dezimal 118) und %00111111 (dezimal 63)
Das ergibt %10110101 (dezimal 181)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | | |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |

Wir beginnen wieder von rechts mit der Addition.

$$0 + 1 = 1$$

$$1 + 1 = 0 + \text{Übertrag } 1$$

$$1 + 1 + \text{Übertrag } 1 = 1 + \text{Übertrag } 1$$

$$0 + 1 + \text{Übertrag } 1 = 0 + \text{Übertrag } 1$$

$$1 + 1 + \text{Übertrag } 1 = 1 + \text{Übertrag } 1$$

$$1 + 1 + \text{Übertrag } 1 = 1 + \text{Übertrag } 1$$

$$1 + \text{Übertrag } 1 = 0 + \text{Übertrag } 1$$

$$0 + 0 + \text{Übertrag } 1 = 1$$

So sieht der zugehörige Assembler-Code aus, als Ergebnis ergibt sich im Akkumulator der Wert \$B5 (dezimal 181 bzw. wie oben in der Tabelle gelb markiert binär %10110101)

```

.A 1500
1500 18      CLC
1501 A9 76    LDA #76
1503 69 3F    ADC #3F
1505 00      BRK
1506 F
,1500 18      CLC
,1501 A9 76    LDA #76
,1503 69 3F    ADC #3F
,1505 00      BRK
-----
.G 1500
PC   SR   AC   XR   YR   SP   NU-BDIZC
;1506 F0  B5  00  00  F6  11110000
.■

```

Beispiel 3

Addieren wir nun die beiden Binärzahlen %01100100 (dezimal 100) und %11001000 (dezimal 200). Das ergibt dezimal 300. Hier ergibt sich jedoch das Problem, dass dieser Wert größer als 255 ist, also nicht mehr mit 8 Bits (also einem Byte) dargestellt werden kann.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| 1 | | | | | | | |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |

Wie bisher beginnen wir wieder von rechts mit der Addition.

$$0 + 0 = 0$$

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$0 + 1 = 1$$

$$0 + 0 = 0$$

$$1 + 0 = 1$$

$$1 + 1 = 0 + \text{Übertrag } 1$$

$$0 + 1 + \text{Übertrag } 1 = 0 + \text{Übertrag } 1$$

So sieht der zugehörige Assembler-Code aus, als Ergebnis ergibt sich im Akkumulator der Wert \$2C (dezimal 44 bzw. wie oben in der Tabelle gelb markiert binär %00101100), die 44 ergibt sich aus der Differenz 300 – 256, also

```

.A 1500
1500 18 CLC
1501 A9 64 LDA #64
1503 69 C8 ADC #C8
1505 00 BRK
1506 F
,1500 18 CLC
,1501 A9 64 LDA #64
,1503 69 C8 ADC #C8
,1505 00 BRK
-----
.G 1500

PC   SR   AC   XR   YR   SP   NU-BDIZC
;1506 31 2C 00 00 F6 00110001
.■

```

Hier findet in der letzten Stelle ein Übertrag statt. Das bedeutet, dass das Ergebnis größer als 255 ist, sich also nicht mehr mit 8 Bits (also einem Byte) darstellen lässt.

Dies wird dadurch gekennzeichnet, dass im Statusregister das Carry Flag auf 1 gesetzt wird, der Übertrag an der linken Stelle wird also in dieses Flag übertragen.

Beispiel 4

Addieren wir nun die beiden Binärzahlen %11111111 (dezimal 255) und %11111111 (dezimal 255)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

Beginnen wir wieder von rechts mit der Addition.

$$\begin{aligned}
 1 + 1 &= 0 + \text{Übertrag } 1 \\
 1 + 1 + \text{Übertrag } 1 &= 1 + \text{Übertrag } 1 \\
 1 + 1 + \text{Übertrag } 1 &= 1 + \text{Übertrag } 1 \\
 1 + 1 + \text{Übertrag } 1 &= 1 + \text{Übertrag } 1 \\
 1 + 1 + \text{Übertrag } 1 &= 1 + \text{Übertrag } 1 \\
 1 + 1 + \text{Übertrag } 1 &= 1 + \text{Übertrag } 1 \\
 1 + 1 + \text{Übertrag } 1 &= 1 + \text{Übertrag } 1 \\
 1 + 1 + \text{Übertrag } 1 &= 1 + \text{Übertrag } 1
 \end{aligned}$$

So sieht der zugehörige Assembler-Code aus, als Ergebnis ergibt sich im Akkumulator der Wert \$FE (dezimal 254 bzw. wie oben in der Tabelle gelb markiert binär %11111110), die 254 ergibt sich aus der Differenz 510 – 256, also

```

.A 1500
1500 18      CLC
1501 A9 FF    LDA #FF
1503 69 FF    ADC #FF
1505 00       BRK
1506 F
;1500 18      CLC
;1501 A9 FF    LDA #FF
;1503 69 FF    ADC #FF
;1505 00       BRK
-----
.G 1500
PC   SR   AC   XR   YR   SP   NU-BDIZC
;1506 B1 FE 00 00 F6  10110001
.■

```

Auch hier findet in der letzten Stelle wieder ein Übertrag statt, d.h. das Ergebnis ist größer als 255, was auch durch ein gesetztes Carry Flag gekennzeichnet wird.

Kommen wir nun zur Bedeutung des Befehls CLC (Clear Carry)

Seine Aufgabe ist sehr einfach, er setzt das Carry Flag im Statusregister zurück d.h. er setzt es auf den Wert 0.

Warum habe ich den Befehl hier immer eingebaut? Der Grund ist, dass der Befehl ADC das Carry Flag in die Rechnung miteinbezieht.

Spielen wir Beispiel 1 nochmals durch, gehen dieses mal jedoch davon aus, dass das Carry Flag gesetzt ist. Mit dem Befehl SEC (Set Carry) kann man übrigens das Carry Flag setzen.

Dann passiert folgendes: Das gesetzte Carryflag wird als „Übertrag“ zu Beginn unserer Addition übernommen, was dann natürlich zu einem völlig anderem Ergebnis führt.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |
| | 1 | 1 | | | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 |

In Beispiel 1 haben wir das Ergebnis %01001101 (dezimal 77) erhalten.

Hier erhalten wir jedoch das Ergebnis %01001110 (dezimal 78) weil zu Beginn der Addition zusätzlich der Übertrag in das Ergebnis eingeflossen ist.

Hier der zugehörige Assembler-Code mit dem Ergebnis \$4E (dezimal 78) im Akkumulator.

```
.A 1500
1500 38      SEC
1501 A9 3A    LDA #3A
1503 69 13    ADC #13
1505 00      BRK
1506 F
;1500 38      SEC
;1501 A9 3A    LDA #3A
;1503 69 13    ADC #13
;1505 00      BRK
-----
.G 1500
PC  SR  AC  XR  YR  SP  NU-BDIZC
;1506 30 4E 00 00 F6  00110000
.■
```

Sie fragen sich jetzt sicher, wofür das gut sein soll und wo man dieses Setzen / Löschen des Carry Flags in Zusammenhang mit der Addition anwenden kann.

Die Antwort lautet:

Addition von zwei 16 Bit Zahlen

Angenommen wir wollen die Zahlen 1000 und 2000 addieren. Das sind beides 16 Bit Werte, das Ergebnis lautet 3000 und ist ebenfalls wieder ein 16 Bit Wert.

Verwenden wir die Speicherstelle \$1500 für das niederwertige Byte des Ergebnisses und die Speicherstelle \$1501 für das höherwertige Byte des Ergebnisses, denn wie wir bereits wissen, werden 16 Bit Werte in dieser Reihenfolge im Speicher abgelegt.

Geben Sie ab Adresse \$1508 folgendes Programm ein und starten es anschließend mit dem Befehl G 1508

```
.A 1508
1508 18      CLC
1509 A9 E8    LDA #E8
150B 69 D0    ADC #D0
150D 8D 00 15  STA 1500
1510 A9 03    LDA #03
1512 69 07    ADC #07
1514 8D 01 15  STA 1501
1517 00      BRK
1518 F
,1508 18      CLC
,1509 A9 E8    LDA #E8
,150B 69 D0    ADC #D0
,150D 8D 00 15  STA 1500
,1510 A9 03    LDA #03
,1512 69 07    ADC #07
,1514 8D 01 15  STA 1501
,1517 00      BRK
-----
.G 1508
PC   SR   AC   XR   YR   SP   NU-BDIZC
;1518 30 0B 00 00 F6 00110000
.
```

Zeigen Sie nun mit dem Befehl M 1500 1510 den Inhalt dieses Speicherbereichs an.

```
.M 1500 1510
:1500 B8 0B FF FF FF FF 00 00  ....7...
:1508 18 A9 E8 69 D0 8D 00 15  ....7...
.
```

An der Speicherstelle \$1500 sieht man das Byte \$B8 und an der Speicherstelle \$1501 das Byte \$0B.

Dies entspricht dem Wert \$0BB8 oder dezimal 3000, also der Summe aus 1000 und 2000.

Wie funktioniert das Programm?

Der Wert 1000 entspricht der Hexadezimalzahl \$03E8, d.h. das niedwertige Byte ist \$E8 und das höherwertige Byte ist \$03.

Der Wert 2000 entspricht der Hexadezimalzahl \$07D0, d.h. das niedwertige Byte ist \$D0 und das höherwertige Byte ist \$07.

Die Summe aus 1000 und 2000 lautet 3000, was der Hexadezimalzahl \$0BB8 entspricht. Das niedwertige Byte ist \$B8 und das höherwertige Byte ist \$0B.

Als erstes setzen wir das Carry Flag mit dem Befehl CLC zurück, da zu Beginn der Addition ja noch kein Übertrag stattgefunden hat.

Dann addieren wir die beiden niederwertigen Bytes der beiden Zahlen, wir rechnen also:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | | | | | | | |
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |

+ 1 Übertrag

An der letzten Stelle ergibt sich also:

$1 + 1 + 1 = 1 + \text{Übertrag } 1$, wodurch das Carry Flag gesetzt wird

Das Ergebnis %10111000 oder \$B8 schreiben wir in die Speicherstelle \$1500, da dies das niederwertige Byte unserer Summe ist.

Nun addieren wir die höherwertigen Bytes der beiden Zahlen, beachten jedoch, dass es bei der Addition der beiden niederwertigen Bytes einen Übertrag gab, was durch das dunkelblaue Feld gekennzeichnet wird.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| | | | | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

Das Ergebnis lautet %00001011 oder \$0B.

Diesen Wert schreiben wir in die Speicherstelle \$1501, da dies das höherwertige Byte unserer Summe ist.

Nun steht die Summe im Speicher wie wir bereits im obigen Speicherdump ab der Adresse \$1500 gesehen haben.

6.4 Die Darstellung von negativen Zahlen

Wollen wir eine negative Zahl, z.B. die -58, binär darstellen, dann rechnen wir zuerst den positiven Wert (58) in eine Binärzahl um.

Das ergibt: %00111010

Nun bilden wir das sogenannte Einerkomplement. Um das Einerkomplement einer Binärzahl zu erhalten, drehen wir alle Bits um, d.h. aus einer 1 wird eine 0 und aus einer 0 wird eine 1.

Einerkomplement: %11000101

Negative Zahlen werden im sogenannten Zweierkomplement dargestellt. Um das Zweierkomplement zu erhalten, addieren wir 1 zum Einerkomplement.

Das ergibt:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 0 | | 0 | 0 | 0 | 0 | 0 | 1 |
| | | | | | | 1 | |
| 1 | | 0 | 0 | 0 | 1 | 1 | 0 |

Die binäre Darstellung der negativen Zahl -58 lautet also %11000110

Negative Zahlen zeichnen sich dadurch aus, dass das am weitesten links stehende Bit 7 gesetzt ist, so wie es auch hier der Fall ist.

Durch die Reservierung des Bit 7 für die Darstellung des Vorzeichens, stehen nur mehr die Bits 0-6 für den eigentlichen Wert zur Verfügung. Dadurch reduziert sich der Darstellungsbereich von vorzeichenbehafteten Zahlen auf -128 bis + 127.

Ohne Reservierung des Bit 7 für das Vorzeichen liegt der Wertebereich ja im Bereich von 0 bis 255.

Nun könnte man fragen:

Der binäre Wert %11000110 entspricht doch auch dem dezimalen Wert 198 oder?
Woran erkennt die CPU ob wir nun die -58 oder die 198 meinen?

Die Antwort lautet:

Gar nicht, die CPU kennt keine negativen Zahlen in diesem Sinn, für sie gibt es nur den binären Wert %11000110.

Es liegt allein am Programmierer, wie der Wert interpretiert werden soll, ob als negative Zahl -58 oder als 198.

Ist das Ergebnis einer Operation negativ, das am weitesten links stehende Bit hat also den Wert 1, dann wird im Statusregister das sogenannte Negative Flag gesetzt. Es steht im Statusregister am weitesten links, also an der Bit Position 7 und stellt eine Kopie des am weitesten links stehenden Bits des Wertes dar, mit dem man das Register geladen hat.

Hier ein Beispiel:

```
.A 1500
1500 A9 C8      LDA #C8
1502 00          BRK
1503 F
;1500 A9 C8      LDA #C8
;1502 00          BRK
-----
.G 1500
PC  SR  AC  XR  YR  SP  NU-BDIZC
;1503 B0 C8 00 00 F6  10110000
.■
```

Hier wird der Wert \$C8 in den Akkumulator geladen, d.h. der binäre Wert %11001000, das am weitesten links stehende Bit hat also den Wert 1.

Als Ergebnis sieht man, dass nun im Statusregister das Negative Flag gesetzt ist.

```
NU-BDIZC
10110000
```

Das Negative Flag ist das am weitesten links stehende Flag (N)

Sehen Sie sich zum Vergleich dieses Beispiel an:

```
.A 1500
1500 A9 7A      LDA #7A
1502 00          BRK
1503 F
;1500 A9 7A      LDA #7A
;1502 00          BRK
-----
.G 1500
PC  SR  AC  XR  YR  SP  NU-BDIZC
;1503 30 7A 00 00 F6  00110000
.■
```

Hier wird der Wert \$7A in den Akkumulator geladen, d.h. der binäre Wert %01111010, das am weitesten links stehende Bit hat also den Wert 0.

Als Ergebnis sieht man, dass dieses mal das Negative Flag nicht gesetzt ist, also den Wert 0 enthält.

7 Logische Verknüpfungen

7.1 UND-Verknüpfung (AND) und ODER-Verknüpfung (OR)

In BASIC haben Sie sicher schon oft mit dem Schlüsselwort IF und in weiterer Folge mit den Schlüsselwörtern AND und OR gearbeitet, um Bedingungen unterschiedlichster Art zu formulieren, unter denen ein bestimmter Programmteil ausgeführt wird oder eben nicht.

Nachfolgend will ich Ihnen zeigen, wie man solche Abfragen in Assembler umsetzen kann.

Fangen wir ganz einfach mit folgendem BASIC-Programm an:



```
10 X=64
20 IF X=64 THEN PRINT "WAHR": GOTO 40
30 PRINT "FALSCH"
40 END
READY.
RUN
WAHR
READY.
```

In Zeile 20 wird hier geprüft, ob eine Aussage ($X=64$) wahr oder falsch ist.

Wenn sie wahr ist, dann wird der Text „WAHR“ ausgegeben, ansonsten der Text „FALSCH“
In diesem Fall ist sie wahr, denn X hat den Inhalt 64.

Würden wir in Zeile 10 für X einen anderen Wert verwenden, dann wäre die Aussage $X=64$ falsch, wodurch in Zeile 30 dann der Text „FALSCH“ ausgegeben wird.

Die Aussage „ $X=64$ “ in Zeile 20 könnte man mit einem Bit (nennen wir es B) gleichsetzen, das entweder den Wert 1 (wahr) oder 0 (falsch) enthält.

Man kann durch Anwendung des Schlüsselwortes NOT die Prüfung der Aussage auch umkehren.

IF NOT($X=64$) THEN PRINT "WAHR": GOTO 40

Der Ausdruck NOT($X=64$) ist umgekehrt genau dann wahr, wenn X einen Wert ungleich 64 besitzt bzw. falsch wenn X den Wert 64 beinhaltet.

Erweitern wir nun unsere Prüfung.

```

10 X=1
20 IF X<5 OR X>100 THEN PRINT"WAHR":GOTO
40
30 PRINT"FAALSCH"
40 END
READY.
RUN
WAHR
READY.

```

Hier werden nun zwei Aussagen überprüft, nennen wir sie A1 und A2.

Die Aussage A1 lautet „X < 5“ und die Aussage A2 lautet „X > 100“

Auch hier können wir die Aussage A1 mit einem Bit B1 und die Aussage A2 mit einem Bit B2 gleichsetzen.

Ist die Aussage A1 wahr, dann ist B1 = 1, andernfalls 0.

Ist die Aussage A2 wahr, dann ist B2 = 1, andernfalls 0.

Welchen Wert hat B1 in unserem Fall hier? Richtig, den Wert 1, da 1 ja kleiner als 5 ist.

Und welchen Wert hat B2? Richtig, den Wert 0, da 1 ja nicht größer als 100 ist.

Durch das Schlüsselwort OR haben wir es hier mit einer ODER-Verknüpfung zu tun, für die folgende Wahrheitstabelle definiert ist.

| X | B1 | B2 | Ergebnis der ODER-Verknüpfung |
|-----|----|----|-------------------------------|
| 25 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 |
| 120 | 0 | 1 | 1 |
| - | 1 | 1 | 1 |

Die grün markierte Zeile stellt unseren Fall hier dar, die Aussage A1 ist wahr und die Aussage A2 ist falsch.

Der Ausdruck **X<5 OR X>100** stellt das Ergebnis der ODER-Verknüpfung von Aussage A1 und Aussage A2 dar. Er ist immer dann wahr, wenn eines der beiden Bits B1 und B2 den Wert 1 hat, oder beide.

Der letzte Fall kann hier jedoch nicht eintreten, da X nicht kleiner als 5 und gleichzeitig größer als 100 sein kann.

Kommen wir zur nächsten Art der Verknüpfung, der UND-Verknüpfung, welche nachfolgend durch das Schlüsselwort AND dargestellt wird.

```

10 X=12
20 IF X>5 AND X<100 THEN PRINT"WAHR":GOT
0 40
30 PRINT"FAALSCH"
40 END
READY.
RUN
WAHR
READY.

```

Die Aussage A1 lautet „X>5“ und die Aussage A2 lautet „X<100“

Da X gleich 12 ist, ist die Aussage A1 wahr und auch die Aussage A2 wahr. Wenn wir nun wieder die Aussage A1 dem Bit B1 und die Aussage A2 dem Bit B2 gleichsetzen, dann hat hier sowohl B1 als auch B2 den Wert 1, da 12 ja größer als 5 und gleichzeitig aber auch kleiner als 100 ist. Für die UND-Verknüpfung ist folgende Wahrheitstabelle definiert.

| X | B1 | B2 | Ergebnis der UND-Verknüpfung |
|-----|----|----|------------------------------|
| - | 0 | 0 | 0 |
| 121 | 1 | 0 | 0 |
| 2 | 0 | 1 | 0 |
| 12 | 1 | 1 | 1 |

Der Ausdruck **X>5 AND X<100** stellt das Ergebnis der UND-Verknüpfung von Aussage A1 und Aussage A2 dar. Er ist immer nur dann wahr, wenn beide Bits B1 und B2 den Wert 1 haben.

Auch hier würde das Voranstellen des Schlüsselwortes NOT die Abfrage umkehren.

NOT (X>5 AND X<100) ist also genau dann wahr, wenn umgekehrt X nicht zwischen 5 und 100 liegt, also z.B. den Wert 2 oder den Wert 121 beinhaltet.

Neben der ODER- und der UND-Verknüpfung gibt es beispielsweise auch noch die Exklusiv-Oder Verknüpfung (XOR)

Hier ist das Ergebnis der Verknüpfung immer dann wahr, wenn die Bits B1 und B2 verschieden sind.

| B1 | B2 | Ergebnis der XOR-Verknüpfung |
|----|----|------------------------------|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Nun möchte ich Ihnen zeigen, wie man in Assembler prüft, ob eine bestimmte Bedingung erfüllt ist und je nach Ergebnis entsprechende Aufgaben ausführt.

Ebenso wie in unserem BASIC-Programm vorhin soll auch in Assembler der Text „WAHR“ oder „FALSCH“ ausgegeben werden, je nachdem ob der jeweilige Ausdruck wahr oder falsch ist.

Doch in Assembler ist das nicht so einfach, denn wir müssen uns zunächst überlegen, wie wir diese Texte ausgeben wollen und wo die Texte überhaupt herkommen.

Ich habe in diesem Beispiel folgende Lösung umgesetzt:

Wir beginnen an der Adresse \$1500, wobei wir hier jedoch nicht mit unserem Programmcode beginnen, sondern ab dieser Adresse die Texte „WAHR“ und „FALSCH“ im Speicher ablegen.

Starten wir also den SMON und geben den Befehl

M 1500 1520

ein.

```
.M 1500 1520
:1500 00 00 FF FF FF FF 00 00
:1504 00 00 FF FF FF FF 00 00
:1510 00 00 FF FF FF FF 00 00
:1518 00 00 FF FF FF FF 00 00
:.
```

Nun bewegen Sie den Cursor in die erste Zeile des Dumps rechts neben der Adresse \$1500 (im Screenshot ist dort der Wert \$00 zu sehen)

Geben Sie nun nacheinander die Werte ein, welche auf folgendem Screenshot zu sehen sind. Wenn Sie den letzten Wert in der ersten Zeile (\$41) eingegeben haben, drücken Sie die RETURN-Taste, damit die Werte übernommen werden.

Danach bewegen Sie den Cursor auf den ersten Wert in der zweiten Zeile neben der Adresse \$1508 und setzen die Eingabe der Werte fort. Die Eingabe endet mit dem Wert \$00 an der Adresse \$150E.

Die Position des Cursors in dem Screenshot ist jene Stelle, an der Sie die RETURN-Taste drücken müssen, damit auch die Werte in dieser Zeile übernommen werden.

```
.M 1500 1520
:1500 57 41 48 52 0D 00 46 41      WAHR..FA
:1504 4C 53 43 48 0D 00 00 00      LSCH....
:1510 00 00 FF FF FF FF 00 00
:1518 00 00 FF FF FF FF 00 00
:.
```

Bewegen Sie nun den Cursor hinter den Punkt und geben dem Befehl A 1510 ein, denn wir wollen nun ab der Adresse \$1510 folgende zwei Unterprogramme erstellen. Nachdem wir den letzten Assembler-Befehl RTS an der Adresse \$152D eingegeben haben, wechseln wir durch Eingabe von F wieder in den Befehlsmodus.

The screenshot shows assembly code in a text-based editor. The code defines two subroutines, each consisting of multiple assembly instructions. The first subroutine, starting at address \$1500, contains the text "WAHR..FA". The second subroutine, starting at address \$151F, contains the text "FALSCH...". Both subroutines use registers X, Y, and Z, and memory locations like 1500, 151E, and FFD2. The editor interface includes a status bar at the bottom with the text "152E F■".

```
PC  SR  AC  XR  YR  SP    NU-BDIZC
;C00B B0 C2 00 00 F6  10110000
.M 1500 1520
:1500 57 41 48 52 0D 00 46 41  WAHR..FA
:1508 4C 53 43 48 0D 00 00 00  LSCH...
:1510 00 00 FF FF FF FF 00 00  .....
:1518 00 00 FF FF FF FF 00 00  .....
.A 1510
:1510 A2 00      LDX #00
:1512 BD 00 15    LDA 1500,X
:1515 F0 07      BEQ 151E
:1517 20 D2 FF    JSR FFD2
:151A E8          INX
:151B 4C 12 15    JMP 1512
:151E 60          RTS
:151F A2 00      LDX #00
:1521 BD 06 15    LDA 1506,X
:1524 F0 07      BEQ 152D
:1526 20 D2 FF    JSR FFD2
:1529 E8          INX
:152A 4C 21 15    JMP 1521
:152D 60          RTS
152E F■
```

Im Befehlsmodus angekommen, speichern wir das Programm unter dem Namen WAHRFALSCH auf Diskette, denn es stellt die Basis dar, auf der die folgenden Programme aufbauen werden.

The screenshot shows a terminal window with the command ".S\"WAHRFALSCH\" 1500 152E" entered and the message "SAVING WAHRFALSCH" displayed below it. The window has a blue background and white text.

```
.S"WAHRFALSCH" 1500 152E
SAVING WAHRFALSCH
.■
```

Doch nun zur Erklärung des soeben geschriebenen Programms, welches in der aktuellen Form nur aus zwei Unterprogrammen besteht.

Das erste Unterprogramm reicht von der Adresse \$1510 bis zur Adresse \$151E und ist dafür zuständig, den Text „WAHR“ mit anschließendem Zeilensprung auf dem Bildschirm auszugeben.

Das zweite Unterprogramm reicht von Adresse \$151F bis zur Adresse \$152D und ist dafür zuständig, den Text „FALSCH“ mit anschließendem Zeilensprung auf dem Bildschirm auszugeben.

Wie funktionieren die beiden Unterprogramme? Fangen wir mit dem ersten an.

Hier wird das X Register als Schleifenzähler verwendet und mit dem Wert \$00 initialisiert.

Der Text WAHR beginnt an Adresse \$1500 und endet an Adresse \$1505.

Ausgegeben werden jedoch nur die Zeichen bis incl. Adresse \$1504, der Wert \$00 an Adresse \$1505 soll das Ende des Strings markieren.

Adresse \$1500 = Zeichencode für W

Adresse \$1501 = Zeichencode für A

Adresse \$1502 = Zeichencode für H

Adresse \$1503 = Zeichencode für R

Adresse \$1504 = Zeichencode für Zeilensprung (\$0D)

Adresse \$1505 = Endemarkierung, ich habe hier den Wert \$00 gewählt

Mit dem Befehl LDA 1500,X beginnt nun eine Schleife, welche nacheinander die Buchstaben des Textes WAHR in den Akkumulator lädt.

Der Aufruf JSR \$FFD2 bewirkt den Aufruf einer Kernal-Routine, welche auch unter dem Namen CHROUT bekannt ist (siehe auch <https://www.c64-wiki.de/wiki/CHROUT>)

Sie liest den Wert im Akkumulator aus und gibt das Zeichen, welches diesem Code entspricht auf dem Bildschirm aus.

Die Schleife läuft solange bis der Wert \$00 in den Akkumulator geladen wird. Dies ist nun jene 0, welche wir oben am Ende des Strings platziert haben.

Dadurch wird das Zeroflag im Statusregister auf 1 gesetzt und dies bewirkt, dass durch den Befehl BEQ 151E zum Befehl RTS an der Adresse \$151E verzweigt wird, was den Rücksprung aus dem Unterprogramm zum Aufrufer bewirkt.

Das zweite Unterprogramm funktioniert genau gleich, nur dass hier der Text FALSCH mit anschließendem Zeilensprung ausgegeben wird.

Der Text FALSCH beginnt an Adresse \$1506 und endet an Adresse \$150D.

Ausgegeben werden jedoch nur die Zeichen bis incl. Adresse \$150C, der Wert \$00 an Adresse \$150D markiert wieder das Ende des Strings.

Adresse \$1506 = Zeichencode für F

Adresse \$1507 = Zeichencode für A

Adresse \$1508 = Zeichencode für L

Adresse \$1509 = Zeichencode für S

Adresse \$150A = Zeichencode für C

Adresse \$150B = Zeichencode für H

Adresse \$150C = Zeichencode für Zeilensprung (\$0D)

Adresse \$150D = Endemarkierung analog zum erstem Unterprogramm

Zugegeben, man wäre sicher auch mit einem Unterprogramm ausgekommen, da sich die beiden ja nur dadurch unterscheiden, welchen Text sie ausgeben. Doch das soll uns an dieser Stelle nicht weiter stören.

Soweit, so gut. Als nächstes möchte ich Ihnen nun zeigen, wie man in Assembler die elementaren Vergleichsoperationen durchführt (gleich, ungleich, größer gleich, kleiner gleich, größer, kleiner)

Die entsprechenden Programme habe ich bereits auf der Diskette vorbereitet, damit Sie diese nicht selbst eingeben müssen.

Bei der Betrachtung der Programme gehen Sie immer nach demselben Schema vor:

- Laden Sie das Programm im SMON mit dem Befehl L und dem jeweiligen Programmnamen
- Wechseln Sie mit dem Befehl X zu BASIC
- Schreiben Sie mit dem Befehl POKE einen beliebigen Wert in die Speicherstelle 781, dieser Wert wird dann beim Starten des Programms mit SYS in das X Register geschrieben
- Starten Sie das Programm mit dem Befehl SYS 5422. Je nachdem welchen Wert Sie vor dem Start des Programms in die Speicherstelle 781 geschrieben haben und welche Vergleichsoperation angewandt wird, gibt das jeweilige Programme entweder den Text WAHR oder FALSCH aus
- Wechseln Sie mit dem Befehl SYS 49152 wieder zurück zu SMON, um das jeweils nächste Programm laden und ausführen zu können

Spielen wir nun die einzelnen Vergleichsoperationen durch, wobei wir mit der Prüfung auf Gleichheit beginnen wollen.

Prüfung „ist gleich“

Laden Sie das Programm mit dem Namen GLEICH von der Diskette und lassen sich mal zur Veranschaulichung den Assemblercode mit dem Befehl D 152E 153C anzeigen.

The screenshot shows a vintage computer terminal window. At the top, it says 'READY' and 'SYS 49152'. Below that is a series of assembly language instructions. The code starts with a header section containing memory dump information (PC, SR, AC, XR, YR, SP) and assembly labels (';C00B', 'L"GLEICH"', 'SEARCHING FOR GLEICH', 'LOADING'). It then lists several instructions: 'D 152E 153C', '152E E0 05 CPX #05', '1530 F0 06 BEQ 1538', '1532 20 1F 15 JSR 151F', '1535 4C 3B 15 JMP 153B'. A dashed line follows, and then more instructions: '1538 20 10 15 JSR 1510', '153B 60 RTS'. At the bottom right is a small green square icon.

```
READY
SYS 49152

      PC   SR   AC   XR   YR   SP   NU-BDIZC
;C00B B0 C2 00 00 F6 10110000
:L"GLEICH"
SEARCHING FOR GLEICH
LOADING
.D 152E 153C
.152E E0 05    CPX #05
.1530 F0 06    BEQ 1538
.1532 20 1F 15  JSR 151F
.1535 4C 3B 15  JMP 153B
-----
.1538 20 10 15  JSR 1510
.153B 60        RTS
-----
```

Zur Erklärung des Programms kommen wir gleich, spielen wir das Programm zunächst erst mal durch.

Wechseln Sie mit X zu BASIC und geben folgende Befehle ein bzw. betrachten das jeweilige Ergebnis.

The screenshot shows a terminal window on a Commodore 64. The screen displays assembly language code and its execution:

```
;153B 60      RTS
.X
READY.
POKE 781,5
READY.
SYS 5422
WAHR
READY.
POKE 781,2
READY.
SYS 5422
FALSCH
READY.
```

Durch den Befehl POKE 781,5 schreiben wir den Wert 5 in die Speicherstelle 781. Diese Speicherstelle kommt Ihnen sicherlich bereits bekannt vor, oder?

Richtig, über diese Speicherstelle kann man den Wert festlegen, mit dem das X Register vor dem Start eines Maschinenprogramms geladen wird, wenn man es mit dem Befehl SYS startet.

Bevor der erste Befehl unseres Programms ausgeführt wird, steht also im X Register bereits der Wert 5. Das ist auch der Grund, warum wir diesen nicht aus der Speicherstelle in das X Register laden müssen, denn das hat ja bereits der Befehl SYS für uns erledigt.

Kommen wir nun zum ersten Befehl im Programm. Durch den Befehl CPX wird der Inhalt des X Registers mit jenem Wert verglichen, den man dahinter angibt. Das kann ein konstanter Zahlenwert oder aber auch der Inhalt einer Speicheradresse sein. In unseren Beispielen hier verwenden wir durchgängig die erste Variante.

Bei der ersten Ausführung des Programms erhalten wir das Ergebnis WAHR, denn durch den Befehl CPX #05 wird der Inhalt des X Registers mit dem Wert 5 verglichen. Im X Register steht bereits der Wert 5 und daher ist hier die Gleichheit erfüllt.

Bei der zweiten Ausführung des Programms erhalten wir erwartungsgemäß das Ergebnis FALSCH, denn 2 ist nicht gleich 5.

Doch wie funktioniert dieser Vergleich mit dem Befehl CPX?

Die CPU subtrahiert bei dessen Ausführung den übergebenen Wert vom Inhalt des X Registers und je nachdem, ob die Differenz gleich 0, größer als 0 oder kleiner als 0 ist, werden entsprechende Flags gesetzt und man kann ausgehend von deren Status dann den passenden Sprungbefehl einsetzen.

Hier eine Übersicht, welche Flags bei welchem Differenz-Ergebnis gesetzt werden.

| | Carry-Flag | Zero-Flag | Negative-Flag |
|---------------|-------------------|------------------|----------------------|
| Differenz = 0 | 1 | 1 | 0 |
| Differenz > 0 | 1 | 0 | 0 |
| Differenz < 0 | 0 | 0 | 1 |

Ist die Differenz gleich 0, dann sind der Inhalt des X Registers und der Vergleichswert gleich.

Ist die Differenz größer als 0, dann ist der Inhalt des X Registers größer als der Vergleichswert.

Ist die Differenz kleiner als 0, dann ist der Inhalt des X Registers kleiner als der Vergleichswert.

Beim ersten Durchlauf hatten wir im X Register den Wert 5, d.h. durch den Befehl CPX #05 ergibt sich Gleichheit, also eine Differenz von 0 und die drei beteiligten Flags haben den Status entsprechend der ersten Zeile in der obigen Tabelle.

Hier sehen wir, dass das Zero-Flag bei Gleichheit gesetzt ist und in den anderen beiden Fällen nicht. Wenn wir also im Falle von Gleichheit an eine andere Stelle im Programm verzweigen wollen, dann brauchen wir dazu einen Sprungbefehl, welcher den Status des Zero-Flags prüft und den Sprung ausführt, wenn dieses gesetzt ist.

Der Sprungbefehl, den wir hier brauchen würden, ist der Befehl BEQ (Branch on Equal). Sein Gegenspieler ist der Befehl BNE (Branch on Not Equal), welcher einen Sprung bei Ungleichheit bewirkt, also wenn das Zero-Flag nicht gesetzt ist.

Kommen wir nun zur Erklärung des Programms, hier nochmal der Screenshot des Assembler-Codes von vorhin:

```

;C00B B8 C2 00 00 F6 10110000
;L"GLEICH"
SEARCHING FOR GLEICH
LOADING
.D 152E 153C
,152E E8 05 CPX #05
,1530 F8 06 BEQ 1538
,1532 20 1F 15 JSR 151F
,1535 4C 3B 15 JMP 153B
-----
,1538 20 10 15 JSR 1510
,153B 60 RTS
-----
.
```

Durch den Befehl CPX wird der aktuelle Inhalt des X Registers mit dem Wert 5 verglichen.

Der Befehl BEQ prüft, ob das Zero-Flag gesetzt ist und wenn ja, wird zur Adresse \$1538 verzweigt. Dort findet durch den Befehl JSR \$1510 der Aufruf des Unterprogramms ab der Adresse \$1510 statt, welches den Text WAHR ausgibt. Anschließend findet ein Sprung zum Befehl RTS an der Adresse \$153B statt, durch den das Programm beendet wird.

Ist das Zero-Flag jedoch nicht gesetzt, wird durch den Befehl JSR \$151F das Unterprogramm ab der Adresse \$151F aufgerufen, welches den Text FALSCH ausgibt. Anschließend wird das Programm durch den Befehl RTS beendet.

Prüfung „ist ungleich“

Falls Sie sich aktuell noch im BASIC befinden, wechseln Sie nun bitte durch Eingabe von SYS 49152 zurück zu SMON.

Laden Sie das Programm mit dem Namen UNGLEICH von der Diskette und lassen sich mal zur Veranschaulichung den Assemblercode mit dem Befehl D 152E 153C anzeigen.



The screenshot shows the assembly code for the 'ungleich' program. The code is as follows:

```
;C64D 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
:L"UNGLEICH"
SEARCHING FOR UNGLEICH
LOADING
.D 152E 153C
,152E E0 05      CPX #05
,1530 D0 06      BNE 1538
,1532 20 1F 15   JSR 151F
,1535 4C 3B 15   JMP 153B
-----
,1538 20 10 15   JSR 1510
,153B 60          RTS
-----
.■
```

An Adresse \$1530 steht nun nicht mehr der Befehl BEQ, sondern sein Gegenspieler, der Befehl BNE, da wir ja nun im Falle von Ungleichheit verzweigen wollen. Hier wird also im Falle eines nicht gesetzten Zero-Flag zur Adresse \$1538 verzweigt. Ansonsten bestehen in Bezug auf den Ablauf keine Unterschiede zum vorherigen Programm.

Wechseln Sie nun mit X zu BASIC und führen dieselben POKE-Befehle aus, die Sie beim vorherigen Programm verwendet haben.

```
,1538 20 10 15 JSR 1510
,153B 60 RTS
-----
.X
READY.
POKE 781,5

READY.
SYS 5422
FALSCH

READY.
POKE 781,2

READY.
SYS 5422
WAHR

READY.
```

Da wir hier nun auf Ungleichheit prüfen, ergibt sich klarerweise das umgekehrte Ergebnis zur Prüfung auf Gleichheit.

Da 5 nicht ungleich 5 ist, erhalten wir das Ergebnis FALSCH und da 2 ungleich 5 ist, erhalten wir das Ergebnis WAHR.

Prüfung „ist größer oder gleich“

Laden Sie das Programm mit dem Namen GRGLEICH von der Diskette und lassen sich mal zur Veranschaulichung den Assemblercode mit dem Befehl D 152E 153E anzeigen.

```
.L"GRGLEICH"
SEARCHING FOR GRGLEICH
LOADING
.D 152E 153E
,152E E0 05 CPX #05
,1530 F0 08 BEQ 153A
,1532 B0 06 BCS 153A
,1534 20 1F 15 JSR 151F
,1537 4C 3D 15 JMP 153D
-----
,153A 20 10 15 JSR 1510
,153D 60 RTS
-----
.■
```

Bei dieser Prüfung ist die Bedingung entweder dann erfüllt, wenn der Wert im X Register gleich 5 oder größer als 5 ist. Im Vergleich zur Prüfung auf Gleichheit und Ungleichheit ist es also nötig, zwei Bedingungen zu prüfen.

Die bereits bekannte Prüfung auf Gleichheit findet hier mit dem Befehl BEQ statt, welcher im Falle dieser zur Adresse \$153A verzweigt. Dort wird das Unterprogramm aufgerufen, welches den Text WAHR ausgibt und das Programm anschließend durch den Befehl RTS beendet.

Falls keine Gleichheit besteht, wird geprüft, ob der Wert im X Register größer als 5 ist. In diesem Fall ergibt sich im Zuge der Verarbeitung des Befehls CPX eine Differenz > 0 und laut der obigen Tabelle wird das Carryflag gesetzt bzw. sowohl das Zero-Flag als auch das Negative-Flag nicht gesetzt.

Hier kommt nun passenderweise der Befehl BCS (Branch on Carry Set) zum Einsatz. Wenn also das Carryflag gesetzt ist, dann wird durch den Befehl BCS \$153A ebenfalls zur Adresse \$153A verzweigt, an der der Aufruf des Unterprogramms zur Ausgabe von WAHR stattfindet.

Schlägt auch diese Prüfung fehl, der Wert im X Register also kleiner als 5 ist, so wird durch den Befehl JSR \$151F das Unterprogramm zur Ausgabe von FALSCH aufgerufen und anschließend zur Adresse \$153D gesprungen, an der das Programm durch den Befehl RTS beendet wird.

Wechseln Sie analog zu den beiden anderen Programmen durch den Befehl X zu BASIC und testen Sie das Programm mit den unterschiedlichsten Werten.

Wie in folgendem Screenshot zu sehen ist, habe ich die Werte 5, 2 und 10 für meine Tests verwendet.

```
.X
READY.
POKE 781,5
READY.
SYS 5422
WAHR

READY.
POKE 781,2
READY.
SYS 5422
FALSCH

READY.
POKE 781,10
READY.
SYS 5422
WAHR

READY.
```

Im ersten Durchlauf erhalten wir das Ergebnis WAHR, da 5 größer oder gleich 5 ist. Beim zweiten Durchlauf erhalten wir das Ergebnis FALSCH, da 2 nicht größer oder gleich 5 ist.

Der dritte Durchlauf bringt das Ergebnis WAHR, da 10 größer oder gleich 5 ist.

Prüfung „ist kleiner oder gleich“

Laden Sie das Programm mit dem Namen KLGLEICH von der Diskette und lassen sich mal zur Veranschaulichung den Assemblercode mit dem Befehl D 152E 153E anzeigen.

```
.D 152E 153E
,152E E8 05      CPX #05
,1530 F0 08      BEQ 153A
,1532 90 06      BCC 153A
,1534 20 1F 15   JSR 151F
,1537 4C 3D 15   JMP 153D
-----
,153A 20 10 15   JSR 1510
,153D 60          RTS
.
.■
```

Hier kommt anstatt des Befehls BCS der Befehl BCC (Branch on Carry Clear) zum Einsatz. Ist der Wert im X Register kleiner als 5, so ergibt sich im Zuge der Verarbeitung des Befehls CPX eine Differenz < 0 . Laut Tabelle ergibt sich dadurch ein gelöschtes Carry-Flag und daher ist der Befehl BCC hier die richtige Wahl.

Wechseln Sie analog zu den beiden anderen Programmen durch den Befehl X zu BASIC und testen Sie das Programm mit den unterschiedlichsten Werten.

Wie in folgendem Screenshot zu sehen ist, habe ich wiederum die Werte 5, 2 und 10 für meine Tests verwendet.

```
.X
READY.
POKE 781,5

READY.
SYS 5422
WAHR

READY.
POKE 781,2

READY.
SYS 5422
WAHR

READY.
POKE 781,10

READY.
SYS 5422
FALSCH

READY.
```

Im ersten Durchlauf erhalten wir das Ergebnis WAHR, da 5 kleiner oder gleich 5 ist.
Beim zweiten Durchlauf erhalten wir das Ergebnis WAHR, da 2 kleiner oder gleich 5 ist.
Der dritte Durchlauf bringt das Ergebnis FALSCH, da 10 nicht kleiner oder gleich 5 ist.

Prüfung „ist größer“

Laden Sie das Programm mit dem Namen GROESSER von der Diskette und lassen sich mal zur Veranschaulichung den Assemblercode mit dem Befehl D 152E 153E anzeigen.

```
L"GROESSER"
SEARCHING FOR GROESSER
LOADING
.D 152E 153E
,152E E0 05      CPX #05
,1530 F0 08      BEQ 153A
,1532 90 06      BCC 153A
,1534 20 10 15   JSR 1510
,1537 4C 3D 15   JMP 153D
,153A 20 1F 15   JSR 151F
,153D 60          RTS
.
■
```

Bei dieser Prüfung muss zunächst auf Gleichheit geprüft werden, denn wenn Gleichheit besteht, kann die Bedingung „größer“ nicht erfüllt sein. Daher wird hier zuerst mit dem Befehl

BEQ \$153A auf Gleichheit geprüft und im Falle dieser zur Adresse \$153A verzweigt, an der das Unterprogramm zur Ausgabe von FALSCH aufgerufen und das Programm danach durch den Befehl RTS beendet wird.

Falls keine Gleichheit besteht, wird mit dem Befehl BCC fortgesetzt, welcher genau dann zur Adresse \$153A verzweigt, wenn das Carry-Flag nicht gesetzt ist.

Dies ist dann der Fall, wenn sich im Zuge der Ausführung des Befehls CPX eine Differenz < 0 ergibt, der Wert im X Register also kleiner ist als 5. In diesem Fall wird zur Adresse \$153A verzweigt, an der das Unterprogramm zur Ausgabe von FALSCH aufgerufen und das Programm danach durch den Befehl RTS beendet wird.

Falls weder die Bedingung „gleich“ noch „kleiner“ erfüllt ist, dann muss die Bedingung „größer“ erfüllt sein und daher wird mit dem Befehl JSR \$1510 fortgesetzt, der das Unterprogramm zur Ausgabe von WAHR aufruft und danach zum Befehl RTS an der Adresse \$153D springt, wodurch das Programm beendet wird.

Testen Sie nun das Programm analog zu den vorherigen Programmen.

Wie in folgendem Screenshot zu sehen ist, habe ich wiederum die Werte 5, 2 und 10 für meine Tests verwendet.

```
.X
READY.
POKE 781,5
READY.
SYS 5422
FALSCH

READY.
POKE 781,2
READY.
SYS 5422
FALSCH

READY.
POKE 781,10
READY.
SYS 5422
WAHR

READY.
```

Im ersten Durchlauf erhalten wir das Ergebnis FALSCH, da 5 nicht größer als 5 ist.
Beim zweiten Durchlauf erhalten wir das Ergebnis FALSCH, da 2 nicht größer als 5 ist.
Der dritte Durchlauf bringt das Ergebnis WAHR, da 10 größer als 5 ist.

Prüfung „ist kleiner“

Laden Sie das Programm mit dem Namen KLEINER von der Diskette und lassen sich mal zur Veranschaulichung den Assemblercode mit dem Befehl D 152E 153E anzeigen.

The screenshot shows the assembly code for the KLEINER program. The code starts with a label L"KLEINER", followed by a search loop labeled SEARCHING FOR KLEINER. It then loads the program and decodes the assembly instructions:

| Instruction | OpCode | Value | Description |
|----------------|--------|-------|-------------|
| .D 152E 153E | | | |
| ,152E E0 05 | E0 | 05 | CPX #05 |
| ,1530 F0 08 | F0 | 08 | BEQ 153A |
| ,1532 B0 06 | B0 | 06 | BCS 153A |
| ,1534 20 10 15 | 20 | 10 | JSR 1510 |
| ,1537 4C 3D 15 | 4C | 3D | JMP 153D |
| ----- | | | |
| ,153A 20 1F 15 | 20 | 1F | JSR 151F |
| ,153D 60 | 60 | | RTS |
| ----- | | | |
| .■ | | | |

Wie beim vorherigen Beispiel wird hier bei Gleichheit durch den Befehl BEQ zur Adresse \$153A verzweigt und das Unterprogramm zur Ausgabe von FALSCH aufgerufen und danach das Programm durch den Befehl RTS beendet.

Falls keine Gleichheit besteht, wird mit dem Befehl BCS fortgesetzt, welcher genau dann zur Adresse \$153A verzweigt, wenn das Carry-Flag gesetzt ist.

Dies ist dann der Fall, wenn sich im Zuge der Ausführung des Befehls CPX eine Differenz > 0 ergibt, der Wert im X Register also größer ist als 5. In diesem Fall wird zur Adresse \$153A verzweigt, an der das Unterprogramm zur Ausgabe von FALSCH aufgerufen und das Programm danach durch den Befehl RTS beendet wird.

Falls weder die Bedingung „gleich“ noch „größer“ erfüllt ist, dann muss die Bedingung „kleiner“ erfüllt sein und daher wird mit dem Befehl JSR \$1510 fortgesetzt, der das Unterprogramm zur Ausgabe von WAHR aufruft und danach zum Befehl RTS an der Adresse \$153D springt, wodurch das Programm beendet wird.

Testen Sie nun das Programm analog zu den vorherigen Programmen.

Wie in folgendem Screenshot zu sehen ist, habe ich wiederum die Werte 5, 2 und 10 für meine Tests verwendet.

```
.X
READY.
POKE 781,5

READY.
SYS 5422
FALSCH

READY.
POKE 781,2

READY.
SYS 5422
WAHR

READY.
POKE 781,10

READY.
SYS 5422
FALSCH

READY.
```

Im ersten Durchlauf erhalten wir das Ergebnis FALSCH, da 5 nicht kleiner als 5 ist.
Beim zweiten Durchlauf erhalten wir das Ergebnis WAHR, da 2 kleiner als 5 ist.
Der dritte Durchlauf bringt das Ergebnis FALSCH, da 10 nicht kleiner als 5 ist.

Nun wissen wir also, wie wir die elementaren Vergleichsoperationen in Assembler umsetzen können. Oft kommt es jedoch vor, dass mehrere Vergleiche in Form einer ODER- bzw. einer UND-Verknüpfung kombiniert werden sollen.

Hier ein Beispiel für eine ODER-Verknüpfung zweier Bedingungen.

```
10 X=7
20 IF X<5 OR X>100 THEN PRINT "WAHR":GOT
0 40
30 PRINT "FALSCH"
40 END
READY.
```

Ist der Inhalt von X kleiner als 5 oder größer als 100 wird der Text WAHR ausgegeben und andernfalls der Text FALSCH.

Da in diesem Beispiel X den Wert 7 enthält, würde der Text FALSCH ausgegeben werden.

Laden Sie das Programm mit dem Namen KL5ODERGR100 von der Diskette und lassen sich mal zur Veranschaulichung den Assemblercode mit dem Befehl D 152E 1544 anzeigen.

The screenshot shows the assembly code for the program KL5ODERGR100. The code is displayed in a blue terminal window. It starts with a search command and then loads the program. The assembly code is as follows:

```
.L"KL5ODERGR100"
SEARCHING FOR KL5ODERGR100
LOADING
.D 152E 1544
.152E E0 05      CPX #05
.1530 F0 08      BEQ 153A
.1532 B0 06      BCS 153A
.1534 20 10 15   JSR 1510
.1537 4C 43 15   JMP 1543
-----
.153A E0 64      CPX #64
.153C F0 E1      BEQ 151F
.153E 90 DF      BCC 151F
.1540 20 10 15   JSR 1510
.1543 60          RTS
.■
```

Zuerst müssen wir prüfen, ob die erste Bedingung X kleiner 5 erfüllt ist. Falls dies nicht der Fall ist, müssen wir mit der Prüfung der zweiten Bedingung X größer 100 weitermachen.

Ist X gleich 5, dann ist die erste Bedingung X kleiner 5 nicht erfüllt und durch den Befehl BEQ \$153A wird zur Adresse \$1513A verzweigt.

Ist X größer 5, dann ist die erste Bedingung X kleiner 5 ebenfalls nicht erfüllt. In diesem Fall ist das Carry-Flag gesetzt und es wird durch den Befehl BCS \$153A zur Adresse \$153A verzweigt.

An der Adresse \$153A beginnt nämlich die Überprüfung der zweiten Bedingung X größer 100. Ist X weder gleich 5 noch kleiner 5, dann ist die erste Bedingung X kleiner 5 erfüllt und es wird mit dem Befehl JSR \$1510 das Unterprogramm zur Ausgabe von WAHR aufgerufen und anschließend zum Befehl RTS an Adresse \$1543 gesprungen, wodurch das Programm beendet wird.

Ob die zweite Bedingung X größer 100 erfüllt ist, müssen wir in diesem Fall gar nicht mehr prüfen, denn das Ergebnis einer ODER-Verknüpfung ist WAHR, sobald eine der beiden Bedingungen erfüllt ist.

Sollte die erste Bedingung jedoch nicht erfüllt sein, dann müssen wir mit der Prüfung der zweiten Bedingung X größer 100 an der Adresse \$153A fortsetzen.

Ist X gleich 100, dann ist die zweite Bedingung X größer 100 nicht erfüllt und durch den Befehl BEQ \$151F wird das Unterprogramm zur Ausgabe von FALSCH aufgerufen.

Ist X kleiner 100, dann ist die erste Bedingung X größer 100 ebenfalls nicht erfüllt. In diesem Fall ist das Carry-Flag nicht gesetzt und es wird durch den Befehl BCC \$151F wird das Unterprogramm zur Ausgabe von FALSCH aufgerufen.

Ist X weder gleich 100 noch kleiner 100, dann ist die zweite Bedingung X größer 100 erfüllt und es wird mit dem Befehl JSR \$1510 das Unterprogramm zur Ausgabe von WAHR aufgerufen und anschließend wird das Programm durch den Befehl RTS beendet.

Kommen wir nun zur UND-Verknüpfung zweier Bedingungen, die durch folgendes Programm dargestellt wird.

```
10 X=7
20 IF X>5 AND X<100 THEN PRINT "WAHR":GO
TO 40
30 PRINT "FALSCH"
40 END
READY.
```

Ist der Inhalt von X größer als 5 und kleiner als 100, so wird der Text WAHR ausgegeben und andernfalls der Text FALSCH.

Da in diesem Beispiel X den Wert 7 enthält, würde der Text WAHR ausgegeben werden.

Doch wie sieht das entsprechende Assembler-Programm aus?

Laden Sie das Programm mit dem Namen GR5UNDKL100 von der Diskette und lassen sich mal zur Veranschaulichung den Assemblercode mit dem Befehl D 152E 1544 anzeigen.

```
,00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
:L"GR5UNDKL100"
SEARCHING FOR GR5UNDKL100
LOADING
.D 152E 1544
,152E E0 05      CPX #05
,1530 F0 0E      BEQ 1540
,1532 90 0C      BCC 1540
,1534 E0 64      CPX #64
,1536 F0 08      BEQ 1540
,1538 B0 06      BCS 1540
,153A 20 10 15    JSR 1510
,153D 4C 43 15    JMP 1543
-----
,1540 20 1F 15    JSR 151F
,1543 60          RTS
-----
.■
```

Kommen wir ohne Umschweife zur Erklärung des Programms.

Als erstes wird geprüft, ob X größer als 5 ist.

Falls X gleich 5 ist, dann wird durch den Befehl BEQ \$1540 zur Adresse \$1540 verzweigt => die Bedingung X größer als 5 ist nicht erfüllt, d.h. es wird FALSCH ausgegeben und das Programm durch den Befehl RTS beendet.

Falls das Carry-Flag nicht gesetzt ist, X also kleiner als 5 ist, dann wird mit dem Befehl BCC \$1540 ebenfalls zur Adresse \$1540 verzweigt und FALSCH ausgegeben, da die Bedingung X größer als 5 nicht erfüllt ist. Anschließend wird das Programm durch den Befehl RTS beendet.

Andernfalls ist die Bedingung X größer 5 erfüllt und es kann geprüft werden, ob auch die zweite Bedingung X kleiner als 100 erfüllt ist.

Falls X gleich 100 ist, dann wird durch den Befehl BEQ \$1540 zur Adresse \$1540 verzweigt => die Bedingung X kleiner als 100 ist nicht erfüllt, d.h. es wird FALSCH ausgegeben und das Programm durch den Befehl RTS beendet.

Falls das Carry-Flag gesetzt ist, X also größer als 100 ist, dann wird mit dem Befehl BCS \$1540 ebenfalls zur Adresse \$1540 verzweigt und FALSCH ausgegeben, da die Bedingung X kleiner als 100 nicht erfüllt ist. Anschließend wird das Programm durch den Befehl RTS beendet.

Andernfalls ist auch die Bedingung X kleiner 100 erfüllt und daher wird mit dem Befehl JSR \$1510 WAHR ausgegeben. Danach wird durch den Befehl JMP \$1543 zum Befehl RTS gesprungen, wodurch das Programm beendet wird.

Nun wollen wir das Programm natürlich auch ausprobieren. Wechseln Sie dazu mit dem Befehl X zu BASIC und starten Sie Durchläufe mit den unterschiedlichsten Werten.

Bei allen Werten, die größer als 5 und kleiner als 100 sind, müsste die Ausgabe WAHR und in allen anderen Fällen FALSCH lauten. Hier einige Beispiele:

```
|POKE 781,5:SYS 5422
FALSCH

READY.
POKE 781,2:SYS 5422
FALSCH

READY.
POKE 781,6:SYS 5422
WAHR

READY.
POKE 781,99:SYS 5422
WAHR

READY.
POKE 781,100:SYS 5422
FALSCH

READY.
POKE 781,101:SYS 5422
FALSCH

READY.
```

7.2 Bitweises Verknüpfen von Bytes

In diesem Kapitel möchte ich Ihnen zeigen, wie man zwei Bytes durch eine ODER -, UND - oder eine EXKLUSIV ODER - Operation miteinander verknüpfen kann. Dabei werden jeweils die zwei Bits an derselben Position miteinander verknüpft, d.h. Bit 7 des ersten Bytes wird mit Bit 7 des zweiten Bytes, Bit 6 des ersten Bytes mit Bit 6 des zweiten Bytes usw. verknüpft.

Befassen wir uns zunächst nochmals mit den Operationen an sich und wiederholen die Regeln, nach denen die Verknüpfungen erfolgen.

Hinweis:

Unter einem gesetzten Bit versteht man ein Bit mit dem Inhalt 1 und unter einem nicht gesetzten Bit ein Bit mit dem Inhalt 0. Letzteres wird je nach Zusammenhang aber auch zurückgesetztes Bit genannt.

7.2.1 ODER - Verknüpfung (OR)

Das Ergebnisbit einer ODER - Verknüpfung zweier Bits ist immer dann gesetzt, wenn eines der beiden Bits oder beide Bits gesetzt sind. Sind beide Bits nicht gesetzt, dann ist auch das Ergebnisbit nicht gesetzt.

Die Fälle, in denen das Ergebnisbit gesetzt ist, sind nachfolgend grün markiert.

| Bit 1 | Bit 2 | Ergebnisbit |
|-------|-------|-------------|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| | | |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 1 | 1 |

Verknüpfen wir als Übung die beiden dezimalen Werte 157 und 201 durch eine ODER - Verknüpfung. Folgende Tabelle zeigt diese beiden Werte in dezimaler, binärer und hexadezimaler Form.

| Dezimal | Binär | hexadezimal |
|---------|-----------|-------------|
| 157 | %10011101 | \$9D |
| 201 | %11001001 | \$C9 |

Führen wir nun die ODER - Verknüpfung durch.

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|---|---|---|---|---|---|---|---|
| Byte 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| Byte 2 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| Ergebnis | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |

Wie wir sehen, ist das jeweilige Bit im Ergebnisbyte immer dann gesetzt wenn eines der beiden Bits gesetzt ist. Dies ist an den Positionen 6, 4 und 2 der Fall.

Aber auch für die Positionen 7, 3 und 0 trifft dies zu. Hier sind beide Bits gesetzt und auch dies führt natürlich zu einem gesetzten Bit im Ergebnisbyte, da die Bedingung, dass eines der beiden Bits gesetzt sein muss, ja ebenfalls erfüllt ist.

Die Bits an den Positionen 5 und 1 sind nicht gesetzt und dies führt zu einem ebenfalls nicht gesetztem Bit im Ergebnisbyte.

7.2.2 UND - Verknüpfung (AND)

Das Ergebnisbit einer UND - Verknüpfung zweier Bits ist nur dann gesetzt, wenn beide Bits gesetzt sind. Sobald eines der beiden Bits nicht gesetzt ist, ist auch das Ergebnisbit nicht gesetzt.

Der Fall, in denen das Ergebnisbit gesetzt ist, ist nachfolgend grün markiert.

| Bit 1 | Bit 2 | Ergebnisbit |
|-------|-------|-------------|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Verknüpfen wir als Übung wieder die beiden dezimalen Werte 157 und 201 von vorhin miteinander, nur dieses mal durch eine UND - Verknüpfung. Folgende Tabelle zeigt wiederum diese beiden Werte in dezimaler, binärer und hexadezimaler Form.

| Dezimal | Binär | hexadezimal |
|----------------|--------------|--------------------|
| 157 | %10011101 | \$9D |
| 201 | %11001001 | \$C9 |

Führen wir nun die UND - Verknüpfung durch.

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| Byte 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| Byte 2 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| Ergebnis | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |

Wie wir sehen, ist das jeweilige Bit im Ergebnisbyte immer nur dann gesetzt, wenn beide Bits gesetzt sind. Dies ist an den Positionen 7, 3 und 0 der Fall.

An den übrigen Positionen ist immer eines der beiden Bits nicht gesetzt und dies führt zu einem nicht gesetztem Bit im Ergebnisbyte. An den Positionen 5 und 1 sind sogar beide Bits nicht gesetzt und natürlich führt auch dies zu einem nicht gesetztem Bit im Ergebnisbyte, da die Bedingung, dass eines der beiden Bits nicht gesetzt ist, ja ebenfalls erfüllt ist.

7.2.3 EXKLUSIV ODER – Verknüpfung (XOR)

Das Ergebnisbit einer EXKLUSIV ODER - Verknüpfung zweier Bits ist nur dann gesetzt, wenn beide Bits verschieden sind, d.h. eines der beiden gesetzt und das andere nicht gesetzt ist. Sobald beide Bits gleich sind, ist das Ergebnisbit nicht gesetzt.

Die Fälle, in denen das Ergebnisbit gesetzt ist, sind nachfolgend grün markiert.

| Bit 1 | Bit 2 | Ergebnisbit |
|--------------|--------------|--------------------|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Bleiben wir bei den beiden vorherigen Beispielwerten und verknüpfen wir als Übung die beiden dezimalen Werte 157 und 201 miteinander, nur dieses mal durch eine EXKLUSIV ODER - Verknüpfung. Folgende Tabelle zeigt wiederum diese beiden Werte in dezimaler, binärer und hexadezimaler Form.

| Dezimal | Binär | hexadezimal |
|----------------|--------------|--------------------|
| 157 | %10011101 | \$9D |
| 201 | %11001001 | \$C9 |

Führen wir nun die EXKLUSIV ODER - Verknüpfung durch.

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|---|---|---|---|---|---|---|---|
| Byte 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| Byte 2 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 |
| Ergebnis | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

Wie wir sehen, ist das jeweilige Bit im Ergebnisbyte immer nur dann gesetzt, wenn beide Bits verschieden sind. Dies ist an den Positionen 6, 4 und 2 der Fall.

An den übrigen Positionen sind die beiden Bits gleich und dies führt zu einem nicht gesetztem Bit im Ergebnisbyte.

Nun werden Sie sich eventuell fragen, wozu man diese Verknüpfungen benötigt.

Diese Operationen brauchen wir beispielsweise immer dann, wenn wir gezielt eines oder mehrere Bits in einem Byte setzen oder zurücksetzen wollen.

Vor allem im Kapitel über Sprites werden wir regen Gebrauch davon machen.

Nachfolgend werden wir uns, ausgestattet mit dem soeben erworbenen Wissen, daher mit dem Setzen und Zurücksetzen von Bits beschäftigen.

Stellen Sie sich für die nachfolgenden Überlegungen ein Byte als eine Reihe von acht Lampen vor, wobei jede Lampe einem Bit entspricht. Ist die Lampe eingeschaltet, entspricht das einem gesetzten Bit. Ist die Lampe ausgeschaltet, entspricht das einem zurückgesetzten Bit.

Die Lampen sind entsprechend ihrer Position von rechts beginnend von 0 bis 7 durchnummeriert und ich möchte dieses Byte in den nachfolgenden Ausführungen als Lampenbyte bezeichnen.

7.2.4 Setzen von Bits

Dazu ist es nur nötig, sich folgende Frage zu stellen:

Wann ist ein Bit im Ergebnisbyte einer ODER - Verknüpfung zweier Bytes gesetzt?

Richtig, wenn eines der beiden korrespondierenden Bits gesetzt ist oder auch beide korrespondierenden Bits gesetzt sind. Mit korrespondierenden Bits sind Bits an derselben Position gemeint.

Angenommen, wir möchten aus einem beliebigen Byte, in dem Bit 6 nicht gesetzt ist, ein neues Byte bilden, dessen Unterschied zum vorherigen Byte nur darin besteht, dass Bit 6 gesetzt ist.

Wie erreichen wir das? Richtig, in dem wir eine ODER - Verknüpfung zwischen dem vorherigen Byte und einem Byte durchführen, welches ausschließlich an Position 6 ein gesetztes Bit enthält.

Nehmen wir als Beispiel das Byte 185, dies entspricht dem binären Wert %10111001 bzw. dem hexadezimalem Wert \$B9. Das Bit an der Position 6 ist nicht gesetzt wie man sieht.

Nun wollen wir aus diesem Byte ein neues Byte bilden, in dem Bit 6 gesetzt ist, das aber ansonsten identisch zum vorherigen Byte ist.

Unser Ergebnisbyte soll also lauten: %11111001

Führen wir also nun Bit für Bit die ODER - Verknüpfung der beiden Bytes durch.

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

Im Ergebnisbyte ist nun Bit 6 gesetzt, die anderen Bits sind identisch zu jenen im vorherigen Byte.

Warum bleiben die anderen Bits unverändert?

Falls ein Bit im vorherigen Byte gesetzt ist, wird es durch die ODER - Verknüpfung mit dem nicht gesetzten Bit im zweiten Byte im Ergebnisbyte ebenfalls gesetzt sein, denn die ODER – Verknüpfung eines gesetzten Bits mit einem nicht gesetzten Bit ergibt ein gesetztes Bit.

Dies ist hier an den Positionen 7, 5, 4, 3 und 0 der Fall.

Falls ein Bit im vorherigen Byte nicht gesetzt ist, wird es durch die ODER - Verknüpfung mit dem nicht gesetzten Bit im zweiten Byte im Ergebnisbyte ebenfalls nicht gesetzt sein, denn die ODER-Verknüpfung zweier nicht gesetzter Bits ergibt ebenfalls ein nicht gesetztes Bit.

Dies ist hier an den Positionen 2 und 1 der Fall.

Das Bit an der Position 6 wird im Ergebnisbyte unabhängig von seinem Status im vorherigen Byte auf jeden Fall gesetzt sein, da das Bit 6 im zweiten Byte ja gesetzt ist und sich dadurch, gemäß den Regeln der ODER – Verknüpfung, immer ein gesetztes Bit im Ergebnisbyte ergibt.

Sie erinnern sich: Ein gesetztes Bit ergibt sich bei der ODER - Verknüpfung immer dann, wenn eines der beiden Bits gesetzt ist. Und genau dies ist hier ja durch das gesetzte Bit 6 im zweiten Byte der Fall.

Nun üben wir das ein wenig, indem wir im Lampenbyte gezielt Bits setzen, also Lampen einschalten.

In den nachfolgenden Ausführungen möchte ich das Byte, welches zum Verknüpfen im Zuge der ODER - Verknüpfung verwendet wird, Verknüpfungsbyte nennen.

Beginnen wir ganz einfach - alle Lampen sind ausgeschaltet und wir wollen Lampe 4 einschalten.

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
|---|---|---|---|---|---|---|---|

Lampe 4 ist hier grün markiert und ist wie alle anderen Lampen aktuell noch ausgeschaltet.

Wie können wir Lampe 4 nun einschalten? Dazu müssen wir eine ODER - Verknüpfung zwischen dem Lampenbyte (aktuell %00000000) und einem Byte durchführen, welches an Position 4 ein gesetztes Bit, also eine 1, enthält (%00010000).

Dadurch ergibt sich ein Ergebnisbyte, welches das neue Lampenbyte darstellt. In diesem neuen Lampenbyte ist dann das Bit an Position 4 gesetzt, Lampe 4 ist also eingeschaltet.

Das Verknüpfungsbyte, welches wir für die ODER - Verknüpfung verwenden müssen, wird hier durch die zweite Zeile mit dem türkisfarbenen Feld an Position 4 dargestellt.

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

Die letzte Zeile stellt das Ergebnisbyte der ODER - Verknüpfung, also das neue Lampenbyte, dar und Sie sehen, dass das Bit an Position 4 nun gesetzt, Lampe 4 also eingeschaltet ist.

Das Verknüpfungsbyte, welches wir für die ODER - Verknüpfung verwendet haben, errechnet sich aus der Methode, welche wir im Kapitel über Zahlensysteme kennengelernt haben.

$$\begin{aligned}
 0 * 2^0 &= 0 * 1 = 0 \\
 0 * 2^1 &= 0 * 2 = 0 \\
 0 * 2^2 &= 0 * 4 = 0 \\
 0 * 2^3 &= 0 * 8 = 0 \\
 1 * 2^4 &= 1 * 16 = 16 \\
 0 * 2^5 &= 0 * 32 = 0 \\
 0 * 2^6 &= 0 * 64 = 0 \\
 0 * 2^7 &= 0 * 128 = 0
 \end{aligned}$$

Die Summe dieser Werte lautet $0 + 0 + 0 + 0 + 16 + 0 + 0 + 0$, also 16, was dem binären Wert %00010000 bzw. dem hexadezimalen Wert \$10 entspricht.

Das Lampenbyte hat den Wert %00000000 bzw. 0, da zu Beginn ja alle Lampen ausgeschaltet sind.

Nun verknüpfen wir die einzelnen Bits gemäß den Regeln für die ODER - Verknüpfung und erhalten das Byte in der orange markierten Zeile. Dies ist nun das neue Lampenbyte.

Setzen wir das doch gleich mal in Assembler um. Starten Sie den SMON und geben folgendes Programm ab Adresse \$1500 ein:

```

PC  SR  AC  XR  YR  SP  NU-BDIZC
;C00B  B0  C2  00  00 F6  10110000
.A 1500
1500  A9  00      LDA #00
1502  09  10      ORA #10
1504  00          BRK
1505  F           -
,1500  A9  00      LDA #00
,1502  09  10      ORA #10
,1504  00          BRK
-----
.G 1500

PC  SR  AC  XR  YR  SP  NU-BDIZC
;1505  30  10  00  00 F6  00110000
.■

```

Im ersten Befehl LDA #00 wird der Akkumulator mit dem Startwert des Lampenbytes geladen, also dem Wert %00000000 (dezimal 0), da ja alle Lampen zu Beginn ausgeschaltet sind.

Der nächste Befehl namens ORA ist neu. Er führt eine ODER - Verknüpfung zwischen dem aktuellen Wert im Akkumulator und dem angegebenen Zahlenwert durch. Das Ergebnis wird wiederum im Akkumulator abgelegt.

Durch den Befehl ORA #10 wird eine ODER - Verknüpfung zwischen dem aktuellen Inhalt des Akkumulators (0) und dem Wert durchgeführt, welcher an Position 4 ein gesetztes Bit enthält.

Vorhin haben wir dafür den dezimalen Wert 16 ermittelt und dies entspricht dem hexadezimalen Wert \$10.

Das war's dann auch schon, es folgt nur mehr der Befehl BRK, welcher das Programm beendet.

Durch Eingabe von F gelangen wir wieder in den Befehlsmodus.

Wir starten unser Programm mit den Befehl G 1500 und wie man in der unteren Ausgabe der Registerinhalte sehen kann, enthält der Akkumulator (AC) nun den hexadezimalen Wert \$10.

Der entsprechende Binärwert lautet %00010000, das Bit 4 im Lampenbyte ist also nun gesetzt, d.h. Lampe 4 ist eingeschaltet.

Speichern Sie das Programm auf Diskette:

```

,1505 30 10 00 00 F6 00110000
.S"SLAMPE4AN" 1500 1505
SAVING LAMPE4AN
.■

```

Nun wollen wir zusätzlich die Lampen 1 und 6 einschalten, also die Bits an den Positionen 1 und 6 im Lampenbyte setzen.

Schritt 1:

Wir bilden eine Binärzahl, die an den Positionen 1 und 6 ein gesetztes Bit enthält.

$$\begin{aligned}0 * 2 \text{ hoch } 0 &= 0 * 1 = 0 \\0 * 2 \text{ hoch } 1 &= 1 * 2 = 2 \\0 * 2 \text{ hoch } 2 &= 0 * 4 = 0 \\0 * 2 \text{ hoch } 3 &= 0 * 8 = 0 \\1 * 2 \text{ hoch } 4 &= 0 * 16 = 0 \\0 * 2 \text{ hoch } 5 &= 0 * 32 = 0 \\0 * 2 \text{ hoch } 6 &= 1 * 64 = 64 \\0 * 2 \text{ hoch } 7 &= 0 * 128 = 0\end{aligned}$$

Die Summe dieser Werte lautet $0 + 2 + 0 + 0 + 0 + 64 + 0$, also 66, was dem binären Wert %01000010 bzw. dem hexadezimalen Wert \$42 entspricht.

Schritt 2:

Wir führen eine ODER-Verknüpfung zwischen dem aktuellen Lampenbyte und dieser Binärzahl durch.

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

Die erste Zeile stellt das aktuelle Lampenbyte dar (nur Lampe 4 ist eingeschaltet)

Die zweite Zeile zeigt die soeben gebildete Binärzahl, welche ein gesetztes Bit an den Positionen 1 und 6 enthält (diese sind türkis markiert)

Die dritte Zeile zeigt das Ergebnis der ODER - Verknüpfung, also das neue Lampenbyte, und man sieht, dass nun zusätzlich zu Lampe 4 nun auch Lampe 1 und Lampe 6 eingeschaltet sind.

Um dies im Assemblercode abzubilden, müssen wir anstelle des Befehls BRK nun den Befehl ORA #42 hinzufügen.

Dies erreichen wir durch Eingabe der folgenden Befehle:

```

;1503 30 10 00 00 r6 00110000
.S" LAMPE4AN" 1500 1505
SAVING LAMPE4AN
.A 1504
 1504 09 42      ORA #42
 1506 00          BRK
 1507 F
,1504 09 42      ORA #42
,1506 00          BRK
-----
.D 1500 1507
,1500 A9 00      LDA #00
,1502 09 10      ORA #10
,1504 09 42      ORA #42
,1506 00          BRK
-----
.G 1500
PC   SR  AC  XR  YR  SP    MU-BDIZC
;1507 30 52 00 00 F6  00110000
.■

```

Durch den Befehl A 1504 können wir neue Befehle ab Adresse \$1504 eingeben, also ab jener Adresse, an der bisher der Befehl BRK gestanden hat.

Hier geben wir nun den Befehl ORA #\$42 ein und um das Programm abzuschließen wiederum den Befehl BRK. Durch Eingabe von F gelangen wir wieder in den Befehlsmodus.

Unser Programm reicht im Speicher nun von Adresse \$1500 bis Adresse \$1506. Durch den Befehl D 1500 1507 können wir uns den Programmcode anzeigen lassen.

Anschließend starten wir unser Programm durch Eingabe von G 1500.

Durch die Anzeige der Registerinhalte sehen wir, dass im Akkumulator der hexadezimale Wert \$52 steht, was dem binären Wert %01010010 entspricht. Die Bits an den Positionen 1, 4 und 6 sind nun gesetzt, d.h. die Lampen 1, 4 und 6 sind eingeschaltet.

Speichern Sie das Programm auf Diskette

```

;1501 30 52 00 00 r6 00110000
.S" LAMPE16EIN" 1500 1507
SAVING LAMPE16EIN
.■

```

Als nächstes wollen wir nun alle Lampen einschalten. Und zwar unabhängig davon, ob bereits eine oder mehrere Lampen eingeschaltet sind.

Schritt 1:

Wir bilden eine Binärzahl, die an allen Positionen ein gesetztes Bit enthält. Dies ist nicht schwierig, denn der entsprechende Wert lautet binär %11111111, was dem dezimalen Wert 255 bzw. dem hexadezimalen Wert \$FF entspricht.

Schritt 2:

Wir führen eine ODER - Verknüpfung zwischen dem aktuellen Lampenbyte und dieser Binärzahl durch.

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

Die erste Zeile stellt das aktuelle Lampenbyte dar (Lampe 1, 4 und 6 sind eingeschaltet)

Die zweite Zeile zeigt die soeben gebildete Binärzahl, welche an allen Positionen ein gesetztes Bit enthält (türkis markiert)

Die dritte Zeile zeigt das Ergebnis der ODER - Verknüpfung, also das neue Lampenbyte, und man sieht, dass nun alle Lampen eingeschaltet sind.

Am Status der bereits eingeschalteten Lampen ändert die ODER - Verknüpfung nichts, denn eine ODER-Verknüpfung zwischen zwei gesetzten Bits ergibt ja wiederum ein gesetztes Bit.

Um dies wieder in Assembler abzubilden, geben wir folgende Befehle ein:

```

;1500 00 00 00 00 10110000
;S" LAMPE1GEIN" 1500 1507
SAVING LAMPE1GEIN
.A 1506
 1506 09 FF      ORA #FF
 1508 00          BRK
 1509 F
,1506 09 FF      ORA #FF
,1508 00          BRK
-----
.D 1500 1509
,1500 A9 00      LDA #00
,1502 09 10      ORA #10
,1504 09 42      ORA #42
,1506 09 FF      ORA #FF
,1508 00          BRK
-----
.G 1500
PC   SR   AC   XR   YR   SP   MU-BDIZC
;1509 B0 FF 00 00 F6 10110000
.■

```

Durch den Befehl A 1506 können wir neue Befehle ab Adresse \$1506 eingeben, also ab jener Adresse, an der bisher der Befehl BRK gestanden hat.

Hier geben wir nun den Befehl ORA #\$FF ein und um das Programm abzuschließen wiederum den Befehl BRK. Durch Eingabe von F gelangen wir wieder in den Befehlsmodus.

Unser Programm reicht im Speicher nun von Adresse \$1500 bis Adresse \$1509. Durch den Befehl D 1500 1509 können wir uns den Programmcode anzeigen lassen.

Starten wir nun das Programm durch Eingabe von G 1500.

Durch die Anzeige der Registerinhalte sehen wir, dass im Akkumulator der hexadezimale Wert \$FF steht, was dem binären Wert %1111111 entspricht. Die Bits an allen Positionen sind nun gesetzt, also alle Lampen sind eingeschaltet.

Speichern Sie das Programm auf Diskette:

```

;1500 00 FF 00 00 10110000
;S" ALLELAMPENEIN" 1500 1509
SAVING ALLELAMPENEIN
.■

```

Natürlich hätten wir das Problem auch anders lösen können, indem wir prüfen, welche Lampen noch ausgeschaltet sind und diese dann auf dieselbe Art und Weise wie in den vorherigen Beispielen einschalten. In diesem Fall wären das die Lampen 0, 2, 3, 5 und 7.

Dazu hätten wir wiederum eine Binärzahl bilden müssen, die an diesen Positionen ein gesetztes Bit enthält und mit dieser dann eine ODER-Verknüpfung durchführen müssen.

$$0 * 2 \text{ hoch } 0 = 1 * 1 = 1$$

$$0 * 2 \text{ hoch } 1 = 0 * 2 = 0$$

$$\begin{aligned}
 0 * 2 \text{ hoch } 2 &= 1 * 4 = 4 \\
 0 * 2 \text{ hoch } 3 &= 1 * 8 = 8 \\
 1 * 2 \text{ hoch } 4 &= 0 * 16 = 0 \\
 0 * 2 \text{ hoch } 5 &= 1 * 32 = 32 \\
 0 * 2 \text{ hoch } 6 &= 0 * 64 = 0 \\
 0 * 2 \text{ hoch } 7 &= 1 * 128 = 128
 \end{aligned}$$

Die Summe dieser Werte lautet $1 + 0 + 4 + 8 + 0 + 32 + 0 + 128$, also 173, was dem binären Wert %10101101 bzw. dem hexadezimalen Wert \$AD entspricht.

Nachfolgend wieder die entsprechende Tabelle analog zu den vorherigen Beispielen:

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

Wie man sieht, ist das Ergebnis identisch. Der Vorteil bei der ersten Vorgangsweise ist jedoch der, dass man nicht im Vorhinein wissen muss, welche Lampen ausgeschaltet sind.

Durch die ODER - Verknüpfung mit dem binären Wert %11111111 werden auf einen Schlag alle Bits gesetzt, unabhängig davon, ob sie vorher bereits gesetzt waren oder nicht.

Um dies wieder in Assembler abzubilden, geben wir folgende Befehle ein:

```

;1503 DD FF 00 00 1503 10110000
;S"ALLELAMPENEIN" 1500 1509
SAVING ALLELAMPENEIN
.A 1506
1506 09 AD      ORA #AD
1508 00          BRK
1509 F
,1506 09 AD      ORA #AD
,1508 00          BRK
-----
.D 1500 1509
,1500 A9 00      LDA #00
,1502 09 10      ORA #10
,1504 09 42      ORA #42
,1506 09 AD      ORA #AD
,1508 00          BRK
-----
.G 1500
PC   SR   AC   XR   YR   SP   NU-BDIZC
;1509 B0  FF  00  00  F6  10110000
.■

```

Durch den Befehl A 1506 können wir neue Befehle ab Adresse \$1506 eingeben, also ab jener Adresse, an der bisher der Befehl ORA #FF gestanden hat.

Hier geben wir nun den Befehl ORA #\$AD ein und um das Programm abzuschließen wiederum den Befehl BRK. Durch Eingabe von F gelangen wir wieder in den Befehlsmodus.

Unser Programm reicht im Speicher wiederum von Adresse \$1500 bis Adresse \$1509. Durch den Befehl D 1500 1509 können wir uns den Programmcode anzeigen lassen.

Abschließend starten wir nun das Programm durch Eingabe von G 1500.

Wie man sieht, enthält der Akkumulator nun ebenfalls den Wert \$FF, d.h. alle Bits sind gesetzt bzw. alle Lampen sind eingeschaltet.

7.2.5 Zurücksetzen von Bits

Dazu ist es nur nötig, sich folgende Frage zu stellen:

Wann ist ein Bit im Ergebnisbyte einer UND - Verknüpfung zweier Bytes nicht gesetzt?

Richtig, wenn eines der beiden korrespondierenden Bits nicht gesetzt ist. Mit korrespondierenden Bits sind wiederum Bits an derselben Position gemeint.

Nehmen wir als Beispiel das Byte 185, dies entspricht dem binären Wert %10111001 bzw. dem hexadezimalen Wert \$B9.

Das Bit an der Position 3 ist bei diesem Byte gesetzt wie wir nachfolgend sehen können.

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

Nun wollen wir dieses gesetzte Bit zurücksetzen.

Wie erreichen wir das? Richtig, indem wir eine UND - Verknüpfung zwischen diesem Byte und einem Byte durchführen, welches ausschließlich an Position 3 ein nicht gesetztes Bit enthält, also mit dem Byte %11110111.

Doch wie kommen wir zu diesem Byte?

Wir wissen, dass wir das Bit an Position 3 zurücksetzen wollen. Bilden wir daher zunächst mal ein Byte, welches an Position 3 ein gesetztes Bit enthält, also das Byte %00001000

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

Dies entspricht dem dezimalen Wert 8 bzw. dem hexadezimalen Wert \$08.

Nun bilden wir von diesem Byte das sogenannte Einerkomplement, d.h. wir drehen alle Bits in dem Byte um. Aus einer 1 wird eine 0 und umgekehrt wird aus einer 0 eine 1. Dadurch erhalten wir ein Byte, das an allen Positionen, außer jener die wir zurücksetzen wollen, eine 1 enthält.

Nachfolgend sehen Sie das Einerkomplement des obigen Bytes.

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

Dies entspricht dem dezimalen Wert 247 bzw. Dem hexadezimalen Wert \$F7.

Um nun ein Byte zu erhalten, in dem das Bit an Position 3 zurückgesetzt ist, müssen wir eine UND - Verknüpfung zwischen unserem ursprünglichen Byte (185) und diesem Einerkomplement - Byte durchführen.

| Byte | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|------|---|---|---|---|---|---|---|---|
| 185 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 247 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |

| | | | | | | | | |
|-----------------|---|---|---|---|---|---|---|---|
| Ergebnis | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
|-----------------|---|---|---|---|---|---|---|---|

Wie wir hier sehen, ist im Ergebnisbyte im Unterschied zum ursprünglichen Byte in der ersten Zeile nun das Bit an der Position 3 zurückgesetzt.

Warum bleiben die anderen Bits unverändert?

Falls ein Bit im vorherigen Byte gesetzt ist, wird es durch die UND - Verknüpfung mit dem gesetzten Bit im Verknüpfungsbyte dann im Ergebnisbyte ebenfalls gesetzt sein, denn die UND – Verknüpfung zweier gesetzter Bits ergibt ebenfalls wieder ein gesetztes Bit.

Dies ist hier an den Positionen 7, 5 und 0 der Fall.

Falls ein Bit im vorherigen Byte nicht gesetzt ist, wird es durch die UND - Verknüpfung mit dem gesetzten Bit im Verknüpfungsbyte dann im Ergebnisbyte ebenfalls nicht gesetzt sein, denn die UND – Verknüpfung zwischen einem gesetzten und einem nicht gesetzten Bit ergibt ein nicht gesetztes Bit.

Dies ist hier an den Positionen 6, 2 und 1 der Fall.

Das Bit an der Position 3 wird im Ergebnisbyte unabhängig von seinem Status im vorherigen Byte auf jeden Fall nicht gesetzt sein, da das Bit 3 im Verknüpfungsbyte ja nicht gesetzt ist und sich durch die UND - Verknüpfung dadurch immer ein nicht gesetztes Bit im Ergebnisbyte ergibt.

Sie erinnern sich: Ein nicht gesetztes Bit ergibt sich bei der UND - Verknüpfung immer dann, wenn eines der beiden Bits nicht gesetzt ist. Und genau dies ist hier ja durch das nicht gesetzte Bit 3 der Fall.

Spielen wir das mal anhand eines einfachen Beispiels durch. Angenommen, es sind alle Lampen eingeschaltet und wir wollen Lampe 2 ausschalten.

Dann benötigen wir zuerst ein Byte, welches an Position 2 ein gesetztes Bit enthält (in der folgenden Darstellung türkis markiert)

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

Nun bilden wir das Einerkomplement dieses Bytes, d.h. aus einer 1 wird eine 0 und umgekehrt wird aus einer 0 eine 1.

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

Nun führen wir eine UND - Verknüpfung zwischen dem aktuellen Lampenbyte (%11111111, aktuell sind also alle Lampen eingeschaltet) und dem soeben gebildeten Einerkomplement - Byte durch.

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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |

Wie man hier sieht, ist im Ergebnisbyte, also dem neuen Lampenbyte, nun an Position 2 ein zurückgesetztes Bit zu sehen, d.h. Lampe 2 ist ausgeschaltet. Der Status der anderen Bits ist unverändert geblieben.

Sehen wir uns gleich mal die Umsetzung in Assembler an.

Doch bevor wir damit beginnen, müssen wir uns zunächst ansehen, wie man das Einerkomplement eines Bytes bildet, da der Befehlssatz der CPU im C64 keinen direkten Befehl dafür bietet.

Wir müssen uns daher anders behelfen.

Die Lösung besteht in einer EXKLUSIV ODER - Verknüpfung (XOR) zwischen dem Byte, von dem man das Einerkomplement bilden will, und dem binären Wert %11111111

Versuchen wir doch gleich mal, das Einerkomplement des Bytes %11010011 (entspricht dem dezimalen Wert 211 bzw. dem hexadezimalen Wert \$D3) über diese Methode zu bilden.

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

Wie man hier sieht, haben wir hier nun in der dritten Zeile als Ergebnisbyte das Einerkomplement des Bytes in der ersten Zeile.

Das Einerkomplement lautet %00101100 (entspricht dem dezimalen Wert 44 bzw. dem hexadezimalen Wert \$2C)

Nachfolgend sehen Sie die Umsetzung dieses Beispiels in Assembler:

```

PC  SR  AC  XR  YR  SP  NU-BDIZC
;C00B  B0  C2  00  00  F6  10110000
.A 1500
1500  A9 D3      LDA #D3
1502  49 FF      EOR #FF
1504  00          BRK
1505  F           BRA
,1500  A9 D3      LDA #D3
,1502  49 FF      EOR #FF
,1504  00          BRK
-----
.G 1500
PC  SR  AC  XR  YR  SP  NU-BDIZC
;1505  30  2C  00  00  F6  00110000
.■

```

Hier wird der Akkumulator mit dem Wert \$D3 geladen und durch den Befehl EOR \$FF mit dem Wert \$FF durch die Exklusiv-Oder Verknüpfung miteinander verknüpft. Das Ergebnis, also das Einerkomplement, wird wiederum im Akkumulator abgelegt.

Nach dem Start des Programms über den Befehl G 1500 sieht man, dass der Akkumulator (AC) nun den Wert \$2C enthält. Dies entspricht wie erwartet dem Ergebnisbyte in der obigen Tabelle.

Da wir nun wissen, wie wir das Einerkomplement eines Bytes bilden können, machen wir uns gleich an die Umsetzung einiger Beispielprogramme.

Zu Beginn sind alle Lampen eingeschaltet, d.h. das Lampenbyte entspricht dem Wert %11111111. Beginnen wir ganz einfach und schalten Lampe 2 aus.

Umsetzung in Assembler:

```

PC  SR  AC  XR  YR  SP  NU-BDIZC
;C00B  B8  C2  00  00 F6  10110000
.A 1500
1500  A9  04      LDA #04
1502  49  FF      EOR #FF
1504  29  FF      AND #FF
1506  00          BRK
1507  F
,1500  A9  04      LDA #04
,1502  49  FF      EOR #FF
,1504  29  FF      AND #FF
,1506  00          BRK
-----
.G 1500
PC  SR  AC  XR  YR  SP  NU-BDIZC
;1507  B8  FB  00  00 F6  10110000
.■

```

Im ersten Befehl LDA #04 wird jener Wert in den Akkumulator geladen, welcher an Position 2 ein gesetztes Bit enthält (%00000100, entspricht dezimal 4 bzw. hexadezimal \$04)

Durch den nächsten Befehl EOR #FF findet die EXKLUSIV ODER - Verknüpfung mit dem Wert %11111111 (dezimal 255 bzw. hexadezimal \$FF) statt.

Im Akkumulator steht nun das Ergebnis dieser Verknüpfung, also das gesuchte Einerkomplement %11111011 (dezimal 251 bzw. hexadezimal \$FB)

Durch den Befehl AND #FF findet schließlich noch die UND - Verknüpfung mit dem aktuellen Lampenbyte statt. Da zu Beginn alle Lampen eingeschaltet sind, entspricht dieses dem Wert %11111111 (dezimal 255 bzw. hexadezimal \$FF)

Nach dem Start des Programms mit dem Befehl G 1500 sieht man, dass nun im Akkumulator der Wert \$FB steht. Dies ist das Ergebnis der UND - Verknüpfung bzw. das neue Lampenbyte, in dem das Bit an Position 2 nun zurückgesetzt ist (Lampe 2 ist ausgeschaltet).

Speichern wir nun dieses kleine Programm auf Diskette:

```

;1501 B8 FB 00 00 F6 10110000
;S" LAMPE2AUS" 1500 1507
SAVING LAMPE2AUS
.■

```

Machen wir nun weiter und schalten die Lampen 1, 5 und 7 aus.

Wir benötigen also wieder ein Byte, welches an den Positionen 1, 5 und 7 ein gesetztes Bit enthält (in der folgenden Darstellung wieder türkis markiert)

Der entsprechende Wert lautet %10100010, dezimal 162 bzw. hexadezimal \$A2

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

Nun bilden wir wieder das Einerkomplement dieses Bytes.

Der entsprechende Wert lautet %01011101, dezimal 93 bzw. hexadezimal \$5D

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

Nun führen wir eine UND - Verknüpfung zwischen dem aktuellen Lampenbyte (%11111011, Lampe 2 ist ausgeschaltet) und dem soeben gebildeten Einerkomplement-Byte durch.

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

Wie man sieht, ist im Ergebnisbyte, also dem neuen Lampenbyte, nun auch an den Positionen 1, 5 und 7 ein zurückgesetztes Bit zu sehen, d.h. zusätzlich zu Lampe 2 sind nun auch die Lampen 1, 5 und 7 ausgeschaltet. Der Status der anderen Bits ist unverändert geblieben.

Das Ergebnisbyte, also das neue Lampenbyte, lautet %01011001, dezimal 89 bzw. hexadezimal \$59

Umsetzung in Assembler:

```

; LAMPEZUSS 1500 1501
SAVING LAMPE2AUS
.A 1506
1506 85 FE STA FE
1508 A9 A2 LDA #A2
150A 49 FF EOR #FF
150C 25 FE AND FE
150E 00 BRK
150F F
,1506 85 FE STA FE
,1508 A9 A2 LDA #A2
,150A 49 FF EOR #FF
,150C 25 FE AND FE
,150E 00 BRK
-----
.G 1500
PC SR AC XR YR SP MU-BDIZC
;150F 30 59 00 00 F6 00110000
.■

```

Durch den Befehl A 1506 erweitern wir unser Programm ab der Stelle, an der beim vorherigen Programm der Befehl BRK gestanden hat.

Da das aktuelle Lampenbyte im Akkumulator durch die Befehle zur Bildung des Einerkomplements überschrieben wird, müssen wir dieses irgendwo sichern. Dazu können wir die Speicherstelle \$FE verwenden. Merken Sie sich für's Erste nur, dass dies eine Speicherstelle in der sogenannten Zeropage ist.

Das Sichern des aktuellen Lampenbytes, welches sich aktuell im Akkumulator befindet, wird durch den Befehl STA FE durchgeführt.

Nun können wir mit dem Befehl LDA #A2 den Akkumulator mit jenem Byte laden, welches an den Positionen 1, 5 und 7 ein gesetztes Bit enthält (\$A2) und durch die Anweisung EOR #FF das Einerkomplement dieses Wertes bilden (\$5D).

Erklärungsbedürftig ist der nächste Befehl AND FE. Hier wird der AND - Befehl nicht mit einem Zahlenwert als Parameter, sondern mit einer Speicheradresse als Parameter aufgerufen, d.h. die UND - Verknüpfung findet dieses mit dem Inhalt einer Speicherstelle statt (in diesem Falle ist dies die Speicherstelle \$FE in welche wir zuvor das aktuelle Lampenbyte gesichert haben).

Nach dem Starten des Programms durch den Befehl G 1500 sieht man, dass der Akkumulator nun den Wert \$59 enthält, den wir auch vorher beim manuellen Durchrechnen ermittelt haben.

Speichern wir das Programm auf Diskette:

```

;1501 30 59 00 00 F6 00110000
:S" LAMPE157AUS" 1500 150F
SAVING LAMPE157AUS
.■

```

Schalten wir nun alle Lampen aus.

Wir benötigen also ein Byte, welches an allen Positionen ein gesetztes Bit enthält (in der folgenden Darstellung wieder türkis markiert)

Der entsprechende Wert lautet %11111111, dezimal 255 bzw. hexadezimal \$FF

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

Nun bilden wir wieder das Einerkomplement dieses Bytes.

Der entsprechende Wert lautet %00000000, dezimal 0 bzw. hexadezimal \$00

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

Nun führen wir eine UND - Verknüpfung zwischen dem aktuellen Lampenbyte (%01011001, die Lampen 1, 2, 5 und 7 sind ausgeschaltet) und dem soeben gebildeten Einerkomplement - Byte durch.

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

Wie man sieht, sind nun alle Bits im Ergebnisbyte, also dem neuen Lampenbyte, zurückgesetzt, d.h. alle Lampen sind ausgeschaltet. Die Lampen 0, 3, 4 und 6 waren vor der UND - Verknüpfung eingeschaltet, aber durch die UND - Verknüpfung mit dem zurückgesetzten Bit im Einerkomplement-Byte werden diese auch ausgeschaltet.

Die bereits ausgeschalteten Lampen 1, 2, 5 und 7 bleiben ausgeschaltet, denn durch die UND - Verknüpfung mit dem zurückgesetzten Bit im Einerkomplement-Byte ändert sich nichts an ihrem Status.

Das Ergebnisbyte, also das neue Lampenbyte, lautet %00000000, dezimal 0 bzw. hexadezimal \$00, d.h. alle Lampen sind nun ausgeschaltet.

Umsetzung in Assembler:

```
SAVING LÄMPE157AÜS
.A 150E
150E 85 FE STA FE
1510 A9 FF LDA #FF
1512 49 FF EOR #FF
1514 25 FE AND FE
1516 00 BRK
1517 F
,150E 85 FE STA FE
,1510 A9 FF LDA #FF
,1512 49 FF EOR #FF
,1514 25 FE AND FE
,1516 00 BRK
-----
.G 1500
PC SR AC XR YR SP MU-BDIZC
;1517 32 00 00 00 F6 00110010
.■
```

Durch den Befehl A 150E erweitern wir unser Programm ab der Stelle, an der beim vorherigen Programm der Befehl BRK gestanden hat.

So wie beim vorherigen Programm sichern wir den Inhalt des Akkumulators wieder in die Speicherstelle \$FE.

Nun können wir mit dem Befehl LDA #FF den Akkumulator mit jenem Byte laden, welches an allen Positionen ein gesetztes Bit enthält (\$FF) und durch die Anweisung EOR #FF das Einerkomplement dieses Wertes bilden (\$00).

Durch den Befehl AND FE führen wir wiederum die UND - Verknüpfung zwischen dem Einerkomplement und dem zuvor gesicherten Lampenbyte aus der Speicherstelle \$FE durch.

Nach dem Starten des Programms durch den Befehl G 1500 sieht man, dass der Akkumulator nun den Wert \$00 enthält, den wir auch vorher beim manuellen Durchrechnen ermittelt haben.

Alle Lampen sind also nun ausgeschaltet.

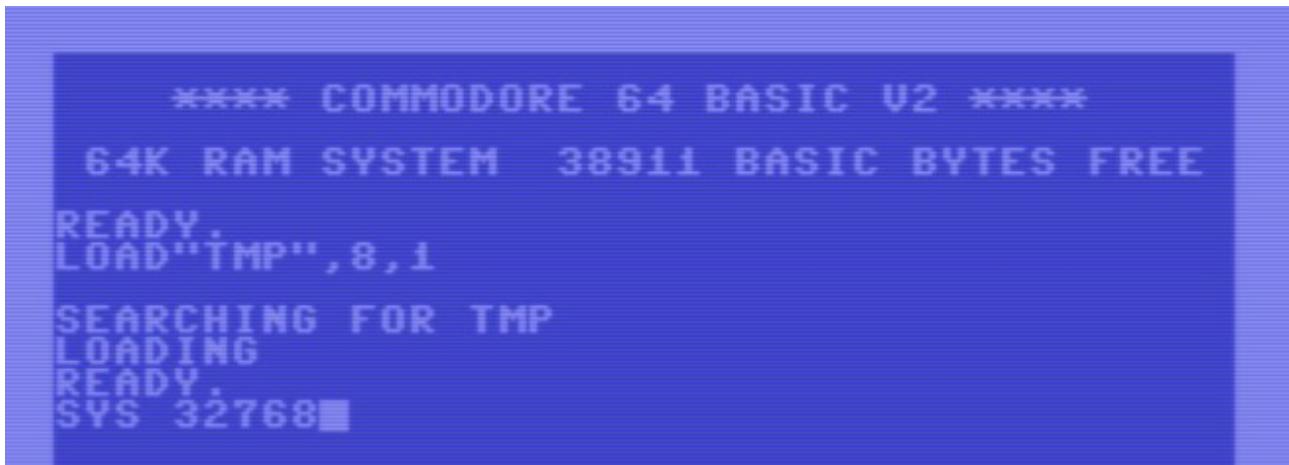
Speichern wir das Programm abschließend noch auf Diskette:

```
;1517 32 00 00 00 F6 00110010
.S"ALLELAMPENAUS" 1500 1517
SAVING ALLELAMPENAUS
.■
```

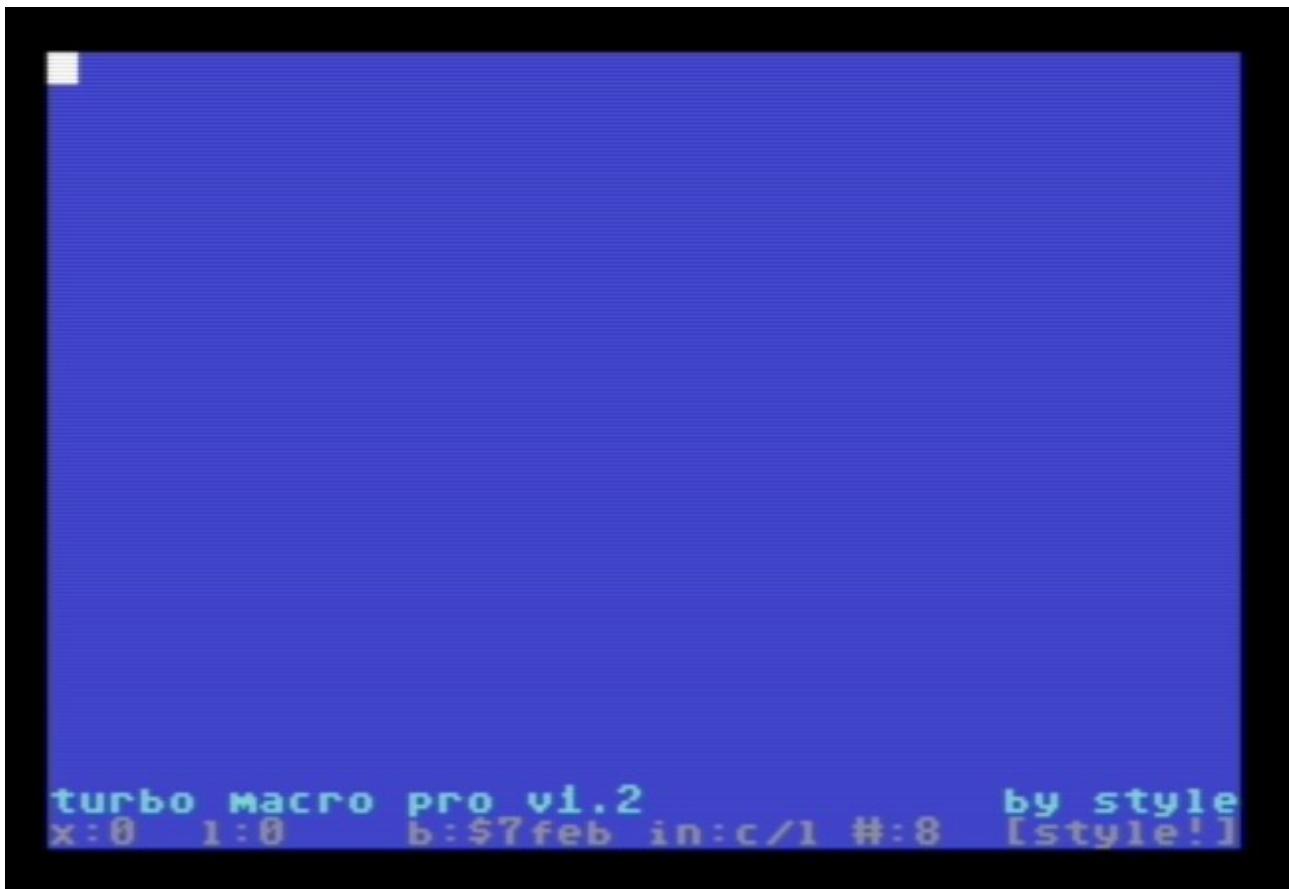
8 Einführung in die Arbeit mit Turbo Macro Pro

Um mit dem Turbo Macro Pro arbeiten zu können, müssen wir ihn natürlich zuerst von der Diskette in den Arbeitsspeicher laden.

Legen Sie also die Diskette mit dem Label TMP ins Laufwerk und laden den Turbo Macro Pro mit der Anweisung LOAD „TMP“,8,1 in den Arbeitsspeicher.

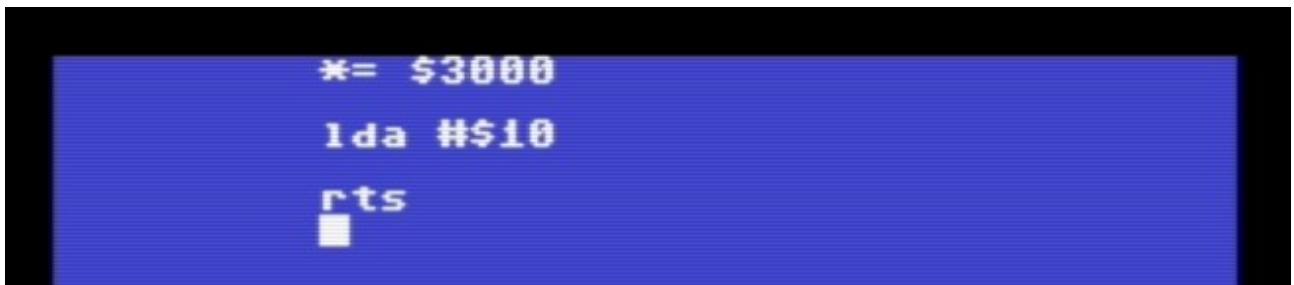


Nachdem Sie den Turbo Macro Pro mit SYS 32768 gestartet haben, sollte sich dieser mit folgender Anzeige präsentieren.



8.1 Eingabe, Assemblierung und Starten eines Assembler-Programms

Beginnen wir also mit einem ganz einfachen Programm und geben folgende Anweisungen ein:



Die Anweisung *= \$3000 legt fest, dass das Programm beginnend bei der Adresse \$3000 im Arbeitsspeicher abgelegt werden soll.

Es folgen zwei einfache Befehle, deren Bedeutung ich Ihnen sicher nicht erklären muss :)

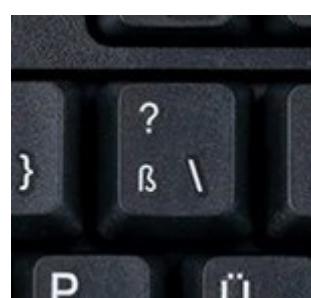
Wie können wir dieses Programm nun starten? Da dieses zum jetzigen Zeitpunkt nur als Text im Editor vorliegt, muss es zunächst einmal assembliert und in Form von Maschinenbefehlen ab der Adresse \$3000 in den Speicher geschrieben werden.

Der Turbo Macro Pro wird über Tastenkombinationen gesteuert, wobei die meisten durch Drücken der Taste, welche den nach links weisenden Pfeil darstellt, eingeleitet werden.

Auf einem echten C64 ist folgende Taste gemeint:

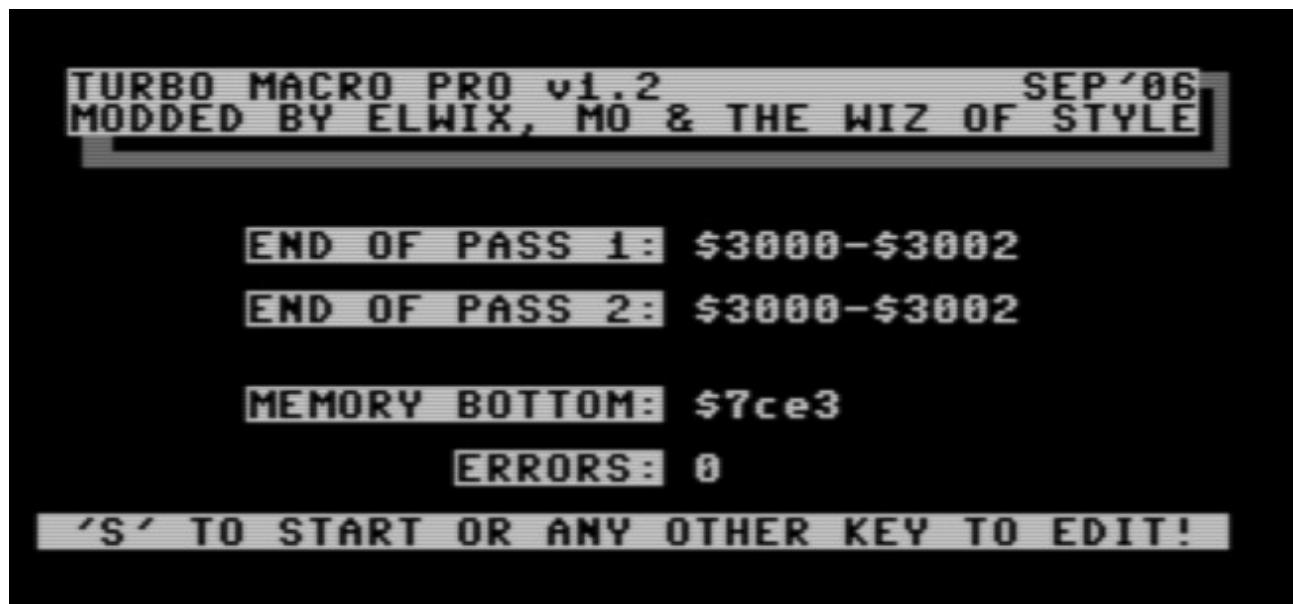


Falls Sie mit einem Emulator, wie z.B. dem VICE arbeiten, dann müssen Sie stattdessen die Taste drücken, auf der das Fragezeichen abgebildet ist:



Um das Programm zu assemblyn, müssen wir die Pfeil-Taste gefolgt von der Taste 3 drücken.

Daraufhin sollte folgende Meldung erscheinen:



Hier wird die Start- und Endadresse unseres Programms angezeigt, es belegt im Speicher also die Adressen von \$3000 bis \$3002.

An den Adressen \$3000 und \$3001 befindet sich der Maschinencode für den Befehl LDA #\$10 und an Adresse \$3002 der Maschinencode für den Befehl RTS.

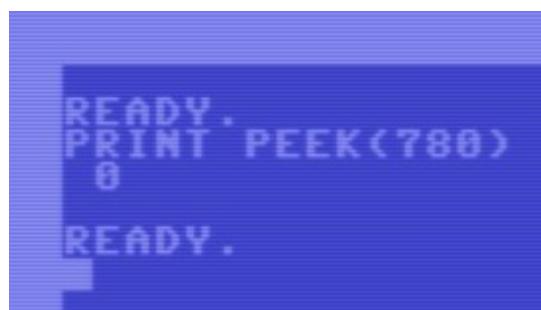
Fehler wurden, wie nicht anders zu erwarten, keine gefunden.

Nun befindet sich unser Programm also als ausführbarer Maschinencode im Speicher. Hier wird angezeigt, dass man das Programm durch Drücken der Taste S starten kann.

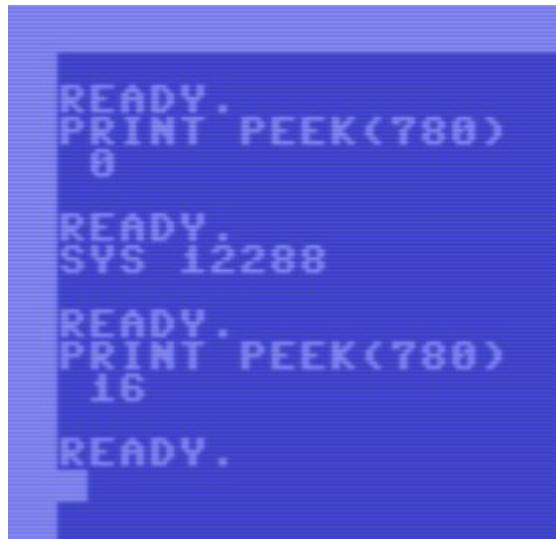
Aus welchen Gründen auch immer, scheint dies jedoch nicht wie erwartet zu funktionieren.

Es findet zwar ein Wechsel zu Basic statt, aber wenn das Programm gestartet worden wäre, müsste der Akkumulator den Wert 16 enthalten.

Ein Auslesen der Speicherstelle 780 zeigt jedoch ein anderes Bild:



Erst wenn das Programm manuell durch Eingabe des Befehls SYS 12288 gestartet wird, stimmt der Inhalt des Akkumulators.



Seltsamerweise hat dies vereinzelt bei anderen Programmen (z.B. bei jenen aus dem später folgenden Kapitel über Sprites) funktioniert und dies sorgte bei mir für etwas Verwirrung.

Daher ziehe ich einen anderen Weg vor, um ein Programm auszuführen.

Ich drücke nicht die Taste S, um das Programm auszuführen, sondern irgendeine andere beliebige Taste, um wieder zum Editor des Turbo Macro Pro zurückzukehren.

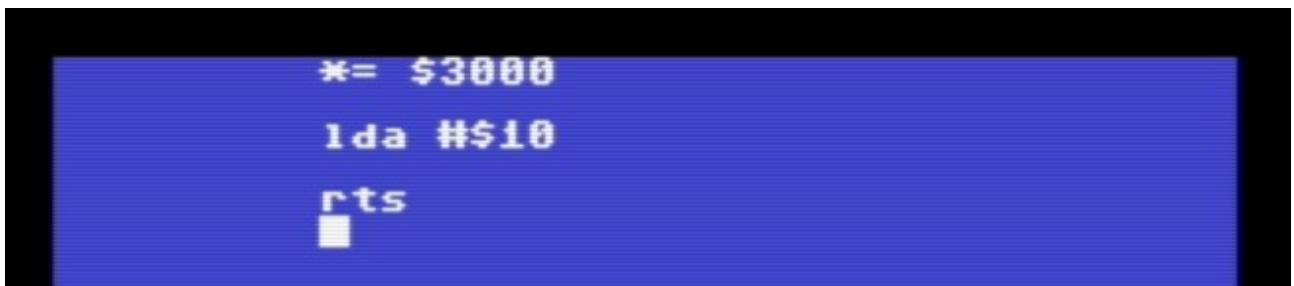
Dort drücke ich die vorhin angesprochene Pfeil-Taste und anschließend die Taste 1.

Dadurch findet ein Wechsel zu Basic statt und man kann das Programm ebenfalls durch Eingabe des entsprechenden SYS-Befehls starten. In diesem Fall hier durch den Befehl SYS 12288 und ich hatte bisher bei keinem einzigen Programm Probleme bei der Ausführung.

Durch Eingabe des Befehls SYS 32768 kann man jederzeit wieder zum Turbo Macro Pro zurückkehren.



Es zeigt sich wieder der Editor mit unserem Assembler-Programm.



```
*= $3000
lda #$10
rts
```

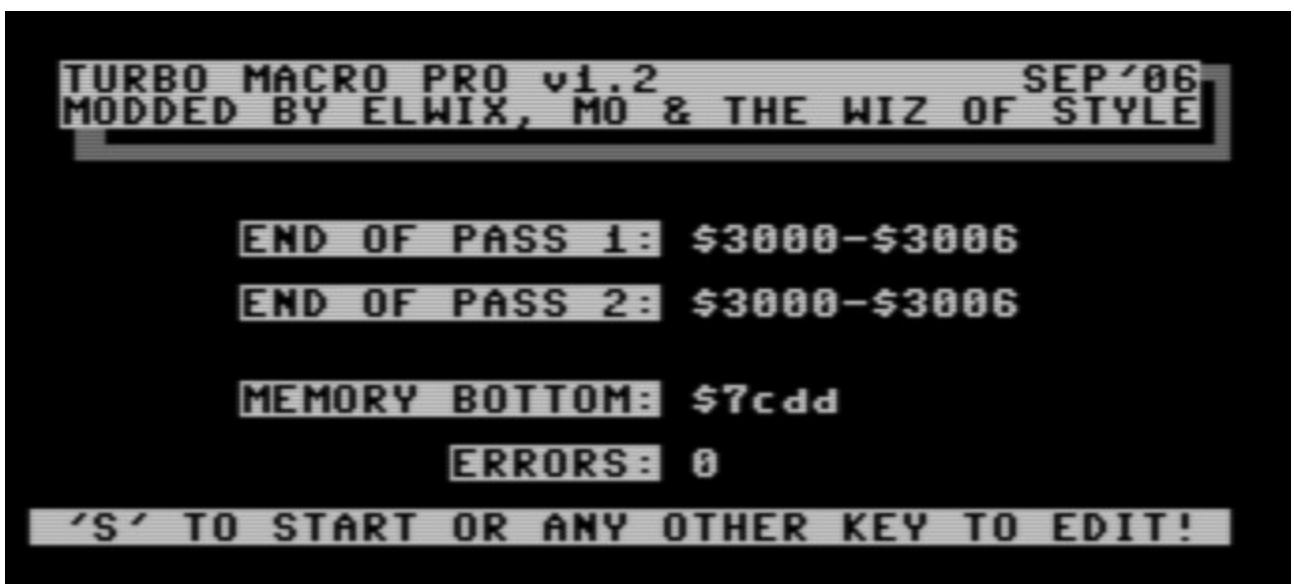
Hier könnten wir nun Änderungen am Programm vornehmen und natürlich werden wir das auch tun :)

Fügen Sie nach dem Befehl LDA #\$10 daher die Befehle LDX #\$20 und LDY #\$30 ein.



```
*= $3000
lda #$10
ldx #$20
ldy #$30
rts
```

Drücken Sie nun die Pfeil-Taste gefolgt von der Taste 3, damit das geänderte Programm assembliert und in den Speicher geschrieben wird.



```
TURBO MACRO PRO v1.2           SEP '06
MODDED BY ELWIX, MO & THE WIZ OF STYLE
```

```
END OF PASS 1: $3000-$3006
END OF PASS 2: $3000-$3006

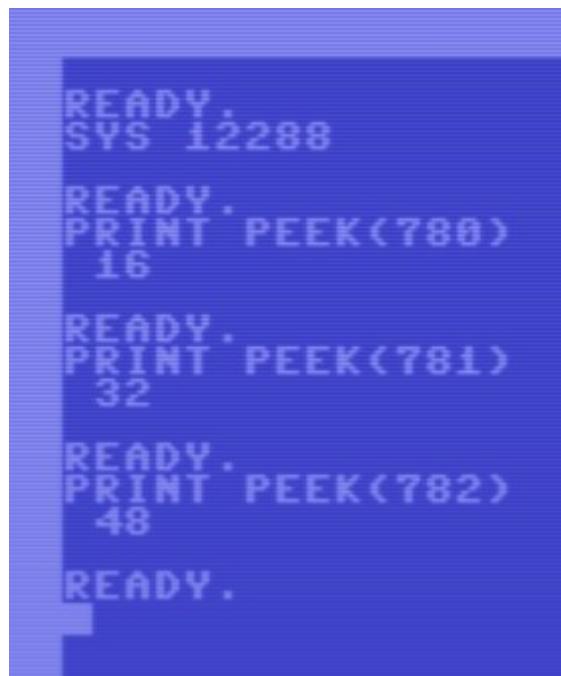
MEMORY BOTTOM: $7cdd
ERRORS: 0

'S' TO START OR ANY OTHER KEY TO EDIT!
```

Hier sieht man, dass sich durch das Hinzufügen der beiden zusätzlichen Befehle die Endadresse des Programms verändert hat. Da sowohl der Befehl LDX #\$20 als auch der Befehl LDY #\$30 zwei Bytes im Speicher belegen, hat sich die Endadresse um vier auf \$3006 erhöht.

Wechseln Sie nun durch Drücken einer beliebigen Taste (außer der Taste S) zurück zum Editor. Dort angekommen drücken Sie die Pfeil-Taste gefolgt von der Taste 1, um ins Basic zu gelangen.

Nun können Sie das Programm wiederum durch Eingabe des Befehls SYS 12288 starten.



```
READY.  
SYS 12288  
  
READY.  
PRINT PEEK(780)  
16  
  
READY.  
PRINT PEEK(781)  
32  
  
READY.  
PRINT PEEK(782)  
48  
  
READY.
```

Das Auslesen der Speicherstellen 780, 781 und 782 beweist, dass der Akkumulator, das X Register und das Y Register genau jene Werte enthalten, welche wir in unserem Programm angegeben haben.

Denn 16 entspricht dem hexadezimalen Wert \$10, 32 dem hexadezimalen Wert \$20 und 48 dem hexadezimalen Wert \$30.

Sehen wir uns nun an, was passiert, wenn der Turbo Macro Pro einen oder mehrere Fehler in unserem Programm gefunden hat.

Wechseln Sie also durch Eingabe von SYS 32768 wieder zum Editor des Turbo Macro Pro und ändern den Befehl LDA in den unbekannten Befehl LDF:



```
*= $3000  
  
ldf #$10  
ldx #$20  
ldy #$30  
█  
rts
```

Wenn Sie das Programm nun durch Drücken der Pfeil-Taste gefolgt von der Taste 3 assemblyn, wird der Turbo Macro Pro folgende Fehlermeldungen anzeigen:

The screenshot shows the assembly output window of Turbo Macro Pro. At the top, it displays the title "TURBO MACRO PRO v1.2 MODDED BY ELWIX, MO & THE WIZ OF STYLE" and the date "SEP '06". The assembly code is as follows:

```
2: bad line
3000 00           ldf #\$10
                  END OF PASS 1: \$3000-\$3005
2: bad line
3000 00           ldf #\$10
                  END OF PASS 2: \$3000-\$3005

MEMORY BOTTOM: \$7cd7
ERRORS: 2

Press Any Key...
```

Durch Drücken einer beliebigen Taste können wir nun zum Editor zurückkehren, um den Fehler zu korrigieren. Machen Sie also die vorhin durchgeführte Änderung wieder rückgängig und assemblyn das Programm durch Drücken der Pfeil-Taste gefolgt von der Taste 3 erneut.

Nun ist wieder alles in Ordnung und das Programm lässt sich auf dem vorhin beschriebenen Wege ausführen.

Fassen wir also die Schritte zusammen, die zur Erstellung und Ausführung eines Assembler-Programms im Turbo Macro Pro nötig sind.

- Assembler-Programm eingeben
- Das Programm durch Drücken der Pfeil-Taste gefolgt von der Taste 3 assemblyn.

Falls der Turbo Macro Pro Fehler im Programm meldet, eine beliebige Taste drücken, um in den Editor zurückzukehren und die Fehler zu korrigieren. Danach muss durch Drücken der Pfeil-Taste gefolgt von der Taste 3 eine erneute Assembly gestartet werden. Der Vorgang muss so oft wiederholt werden, bis vom Turbo Macro Pro keine Fehler mehr gemeldet werden.

- Wenn der Turbo Macro Pro keine Fehler meldet, eine beliebigen Taste außer der Taste S drücken, um wieder in den Editor zurückzukehren
- Durch Drücken der Pfeil-Taste gefolgt von der Taste 1 ins Basic wechseln

- Das Programm durch Eingabe des Befehls SYS gefolgt von der im Assembler-Programm festgelegten Adresse starten (im Falle von \$3000 also 12288, da die Adresse in dezimaler Form angegeben werden muss)

8.2 Speichern eines Assembler-Programms auf Diskette

Bisher existiert unser Assembler-Programm nur als Text im Editor des Turbo Macro Pro. Um das Programm jedoch dauerhaft zur Verfügung zu haben, müssen wir es auf einer Diskette speichern, damit wir es dann später wieder laden, starten oder ggf. auch ändern können.

Um unser Assembler-Programm auf Diskette speichern zu können, müssen wir die Pfeil-Taste gefolgt von der Taste S drücken.

Dadurch erscheint am unteren Rand eine Frage nach einem Dateinamen, unter dem das Assembler-Programm gespeichert werden soll.



```

*= $3000
 lda #$10
 ldx #$20
 ldy #$30
 rts

save file: tmptest1
x:19 1:5 b:$7fdd in:c/l #:8 [style!]

```

Geben Sie hier einen beliebigen Dateinamen, z.B. wie in diesem Fall hier den Namen tmptest1 ein und drücken die Return-Taste.

Wenn alles gut gegangen ist, wird am unteren Rand eine entsprechende Meldung angezeigt:



8.3 Laden eines Assembler-Programms von Diskette

Gehen wir nun zurück zum Anfang, starten unseren C64 oder Emulator neu und laden den Turbo Macro Pro.

Nachdem wir diesen durch Eingabe von SYS 32768 gestartet haben, sehen wir wieder den leeren Editor.

Um das Programm, welches wir vorhin unter dem Namen tmptest1 gespeichert haben, wieder zu laden, drücken Sie die Pfeil-Taste gefolgt von der Taste L.

Dadurch wird am unteren Rand eine Frage nach dem Namen der Datei angezeigt, die geladen werden soll. Geben Sie dort den Namen tmptest1 ein.



Nachdem Sie die Return-Taste gedrückt haben, wird das Programm wieder in den Editor geladen.

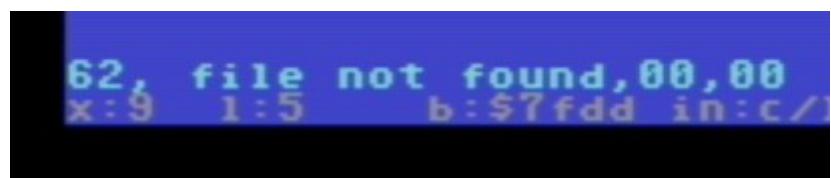
The screenshot shows the Commodore 64 assembly editor. At the top, the assembly code is displayed:

```
*= $3000
lda #$10
ldx #$20
ldy #$30
rts
```

At the bottom of the screen, the status bar displays the message "00, ok, 00, 00" followed by file information: "x:9 1:5 b:\$7fdd in:c/l #:8 [style!]".

Am unteren Rand wird nun eine ok-Meldung angezeigt. Falls beim Laden ein Fehler aufgetreten ist, würde hier eine entsprechende Fehlermeldung angezeigt werden.

Angenommen, Sie wollen eine Datei laden, die nicht existiert, dann würde dies durch folgende Meldung angezeigt werden:



Nachdem wir unser Programm nun wieder geladen haben, können wir es auch wieder assemblieren und starten.

8.4 Überschreiben eines bereits bestehenden Assembler-Programms

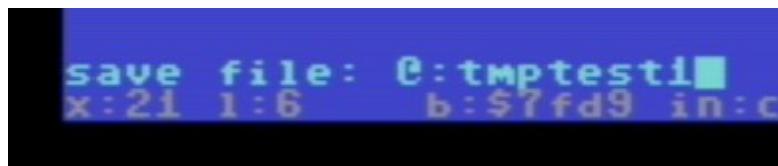
Da wir unser Programm nach dem Laden beliebig verändern können und diese Änderungen natürlich auch in die Datei auf der Diskette übernehmen wollen, gilt es beim erneuten Speichern der Datei eine Besonderheit zu beachten.

Wenn wir Änderungen an unserem Programm vorgenommen haben, müssen wir beim anschließenden Speichern der Datei die Zeichenfolge @: vor dem Dateinamen einfügen. Dadurch wird die bereits bestehende Datei auf der Diskette überschrieben und enthält nun die von uns eventuell vorgenommenen Änderungen.

Fügen Sie also beispielsweise den Befehl STA \$0400 ein:



Wenn Sie nun diese Änderung in die bereits bestehende Datei tmptest1 übernehmen wollen, dann müssen Sie anschließend bei der Abfrage des Dateinamens die Eingabe wie folgt ändern:

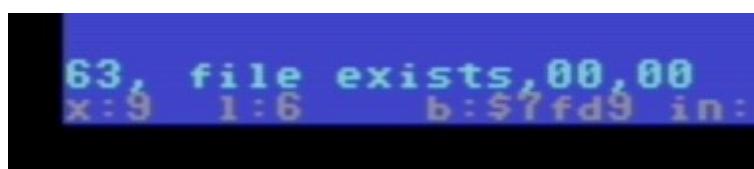


Bewegen Sie nun den Cursor hinter die letzte 0 der Adresse \$0400, welche dem Befehl STA folgt, und entfernen durch mehrmaliges Drücken der Backspace Taste den Befehl STA \$0400 wieder aus dem Programm.

Laden Sie nun die Datei tmptest1 auf die vorhin beschriebene Weise durch Drücken der Pfeil-Taste gefolgt von der Taste L und anschließender Eingabe des Dateinamens tmptest1.

Sie werden sehen, dass der STA Befehl in die Datei übernommen wurde und nun wieder im Editor angezeigt wird.

Wenn Sie die Zeichenfolge @: vor dem Dateinamen nicht angeben, dann erhalten Sie die Meldung, dass die Datei bereits existiert:



Die Änderungen werden in diesem Fall nicht in die bestehende Datei übernommen.

Wichtige Anmerkung:

Egal ob Sie ein völlig neues Programm eingeben oder Änderungen an einem bereits bestehenden Programm vornehmen – die oberste Regel lautet immer: Vor dem Starten des Programms IMMER den aktuellen Stand des Programms auf Diskette speichern!

Gerade bei der Assembler-Programmierung kommt es sehr oft vor, dass es zu Abstürzen des C64 kommt und man dadurch keine Gelegenheit mehr hat, in den Editor des Turbo Macro Pro zurückzukehren und den aktuellen Stand des Programms auf Diskette zu sichern.

In diesem Fall war dann die gesamte Arbeit umsonst und Sie müssen mit der Eingabe von vorne beginnen. Dies kann, je nach Umfang des Programms und der durchgeführten Änderungen, natürlich sehr viel Zeit verschlingen und ziemlich frustrierend sein.

Nun beherrschen Sie bereits die grundlegenden Schritte, die nötig sind, um sinnvoll mit dem Turbo Macro Pro arbeiten zu können.

Es gibt jedoch noch eine ganze Menge weiterer Tastenfunktionen, die Ihnen bei der Arbeit gute Dienste leisten können.

8.5 Erstellen eines eigenständigen Programms aus einem Assembler-Programm

Nachdem man ein Assembler-Programm erstellt und erfolgreich getestet hat, will man im Normalfall daraus natürlich eine Datei erstellen, die man ohne Zuhilfenahme des Turbo Macro Pro von der Diskette laden und ausführen kann.

Der Turbo Macro Pro bietet diese Funktion durch Drücken der Pfeil-Taste gefolgt von der Taste 5 an.

Daraufhin wird am unteren Rand die Frage nach einem Dateinamen angezeigt, unter dem das eigenständige Programm auf der Diskette gespeichert werden soll.

```
*= $3000
lda #$10
ldx #$20
ldy #$30
sta $0400
rts

object file: tmptest1prg
x:24 1:6    b:$7fd9 in:c/1 #:8 [style!]
```

Geben Sie in unserem Fall den Namen tmptest1prg ein und Drücken die Return-Taste.
Daraufhin wird folgende Meldung angezeigt:

```
TURBO MACRO PRO v1.2           SEP '86
MODDED BY ELWIX, MO & THE WIZ OF STYLE
```



```
END OF PASS 1: $3000-$3009
END OF PASS 2: $3000-$3009

MEMORY BOTTOM: $7cd9
ERRORS: 0

Press Any Key...
```

Wenn alles, wie in diesem Fall (0 Errors), gut gegangen ist, wird die Datei tmptest1prg auf der Diskette erstellt.

Starten Sie nun den C64 neu, laden das Programm wie folgt von der Diskette und starten es mit dem Befehl SYS 12288.

```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
LOAD"TMPTEST1PRG",8,1
SEARCHING FOR TMPTEST1PRG
LOADING
READY.
SYS 12288
```

Nach dem Start des Programms wird in der linken, oberen Ecke des Bildschirms der Buchstabe P ausgegeben. Dies wurde durch den neu hinzugefügten Befehl STA \$0400 bewirkt und durch Eingabe von PRINT gefolgt von PEEK's auf die Speicherstellen 780, 781 und 782 zeigt sich, dass der Akkumulator, das X Register und das Y Register die gewünschten Werte beinhalten.

```
P
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
READY.
LOAD"TMPTEST1PRG",8,1
SEARCHING FOR TMPTEST1PRG
LOADING
READY.
SYS 12288

READY.
PRINT PEEK(780),PEEK(781),PEEK(782)
16      32      48

READY.
```

Angenommen, wir würden nun den SMON starten und ab Adresse \$3000 disassemblieren, dann sehen wir, dass unser Programm, wie erwartet, ab der Adresse \$3000 im Speicher zu finden ist.

```

.D 3000
;3000 A9 10    LDA #10
;3002 A2 20    LDX #20
;3004 A0 30    LDY #30
;3006 8D 00 04  STA 0400
;3009 60      RTS
-----.

```

Wir können das Programm im SMON natürlich auch durch Eingabe des Befehls G 3000 starten.

Zuvor müssen wir jedoch den Befehl RTS in den Befehl BRK ändern, damit der SMON nach Beendigung des Programms nicht verlassen wird. Würden wir es beim Befehl RTS belassen, würde ja ein Sprung ins Basic stattfinden.

Diese Änderung können wir natürlich, wie bereits von unserer Arbeit mit SMON bekannt, direkt im SMON durchführen, indem wir den Cursor auf den Befehl RTS bewegen und stattdessen den Befehl BRK eingeben.

Anzeige nach dem Start des Programms:

```

PG 3000
PC  SR  AC  XR  YR  SP  MU-BDIZC
;300A 30 10 20 30 F6  00110000

```

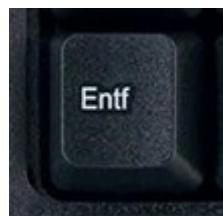
In der linken, oberen Ecke sieht man wieder den Buchstaben P und die Inhalte des Akkumulators (AC), des X Registers (XR) und des Y Registers (YR) entsprechen auch den erwarteten Werten \$10, \$20 und \$30.

8.6 Löschen von einzelnen Zeilen

Wollen Sie eine einzelne Zeile in Ihrem Assembler-Programm löschen, dann bewegen Sie den Cursor in diese Zeile und drücken die Pfeil-Taste gefolgt von der INST/DEL Taste, wenn Sie auf einem echten C64 arbeiten.



Falls Sie im Emulator arbeiten, dann drücken Sie die Taste, auf der das Fragezeichen zu sehen ist, gefolgt von der Taste mit der Aufschrift „Entf“.



Die Zeile, in der sich der Cursor befindet, wird gelöscht und die darunterliegenden Zeilen rücken um eine Zeile nach oben.

8.7 Kommentare

Durch Kommentare wird die Lesbarkeit eines Programms stark verbessert. Auch der Turbo Macro Pro bietet die Möglichkeit, Programme mit Kommentarzeilen zu versehen. Dazu muss man der jeweiligen Zeile ein Semikolon voranstellen.

Angewandt auf unser kleines Programm vom Beginn dieser Einführung, könnte dies dann so aussehen:

The screenshot shows the Turbo Macro Pro v1.2 assembly editor interface. The main window displays the following assembly code:

```
; programm beginnt bei
; adresse $3000
; start von basic aus mit
; sys 12288
*= $3000
; akkumulator mit dem wert
; $10 (dezimal 16) laden
lda #\$10
; programm beenden
rts
■
```

In the bottom status bar, the text "turbo macro pro v1.2" and "by style" is visible, along with other system information: "x:9 1:15 b:\$7f56 in:c/l #:8 [style!]".

Bei solch einfachen Befehlen sind im Normalfall keine Kommentare nötig, aber es sollte hier ja auch nur gezeigt werden, wie man Kommentare in ein Assembler-Programm integrieren kann.

8.8 Einfügen von Trennzeilen

Manchmal nutzt man Kommentarzeilen, um Programmteile optisch voneinander abzugrenzen.

Damit man diese Kommentarzeilen nicht selber, Zeichen für Zeichen, eingeben muss, bietet der TMP ebenfalls eine entsprechende Tastenkombination über die Pfeil-Taste gefolgt von der Taste 2 an.

Ich habe nachfolgend zur Veranschaulichung eine solche Trennzeile nach dem Befehl RTS eingefügt.



8.9 Blockoperationen

Manchmal ist es notwendig, nicht nur eine einzige Zeile, sondern einen ganzen Block, welcher aus mehreren Zeilen besteht, zu löschen, zu kopieren oder zu verschieben. Unabhängig davon, ob man einen Block löschen, kopieren oder verschieben will, muss man klarerweise zunächst den Beginn und das Ende des Blocks festlegen.

Ich habe im Editor folgende fünf Kommentarzeilen eingegeben:

```
; zeile 1
; zeile 2
; zeile 3
; zeile 4
; zeile 5
```

x:9 1:5 b:\$7fbe in:c/l #8 [style!]

Angenommen, wir möchten nun aus den ersten drei Zeilen einen Block bilden.
Als erstes bewegen wir den Cursor auf den Strichpunkt in Zeile 1.

```
■ zeile 1
; zeile 2
; zeile 3
; zeile 4
; zeile 5
```

Dann drücken wir die Pfeil-Taste gefolgt von der Taste M.

Am unteren Rand erscheint nun folgende Abfrage:

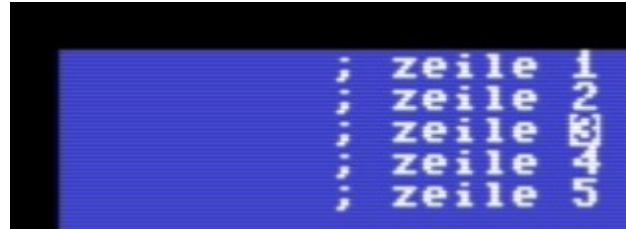
```
set mark (0-9,s,e): █
```

x:20 1:0 b:\$7fbe in

Der Turbo Macro Pro bietet hier durch die Angabe 0-9 offensichtlich die Möglichkeit, mit mehreren Blöcken zu arbeiten. Ich habe diese Möglichkeit jedoch noch nie genutzt und werde daher auch nicht näher darauf eingehen und mich stattdessen auf einen einzigen Block beschränken.

Hier drücken wir die Taste S, wodurch der Beginn des Blocks festgelegt wird.

Nun bewegen wir den Cursor auf die 3 in Zeile 3.



```
; zeile 1  
; zeile 2  
; zeile 3  
; zeile 4  
; zeile 5
```

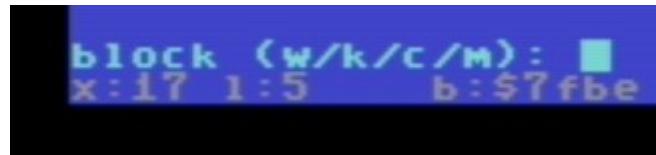
Als nächstes drücken wir wieder die Pfeil-Taste gefolgt von der Taste M. Die obige Abfrage erscheint erneut, nur dieses mal drücken wir die Taste E, wodurch nun auch das Ende des Blocks festgelegt wird.

Diesen Block wollen wir nun direkt unterhalb von Zeile 5 anfügen. Dazu bewegen wir den Cursor direkt unter das Semikolon in Zeile 5.



```
; zeile 1  
; zeile 2  
; zeile 3  
; zeile 4  
; zeile 5
```

Nun drücken wir die Pfeil-Taste gefolgt von der Taste B, woraufhin am unteren Rand folgende Abfrage erscheint:



```
block (w/k/c/m): █  
x:17 1:5 b:$7fbe
```

Hier haben wir die Möglichkeit, auszuwählen, ob wir den Block in eine Datei schreiben, ihn löschen, kopieren oder verschieben wollen. Da wir den Block kopieren wollen, drücken wir hier die Taste C, woraufhin die drei Zeilen kopiert werden.

Inhalt des Editors nach dem Kopiervorgang:



```
; zeile 1  
; zeile 2  
; zeile 3  
; zeile 4  
; zeile 5  
; zeile 1  
; zeile 2  
; zeile 3
```

Als nächstes wollen wir denselben Block, also wiederum die ersten drei Zeilen, verschieben. Dazu legen wir, wie vorhin beschrieben, zunächst den Beginn und das Ende des Blocks fest und bewegen den Cursor wieder direkt unter das Semikolon in Zeile 5.

Nun drücken wir wieder die Pfeil-Taste gefolgt von der Taste B, woraufhin wieder die vorherige Auswahl-Abfrage erscheint.

Dieses mal drücken wir die Taste M, da wir den Block verschieben wollen.

Inhalt des Editors nach dem Verschieben des Blocks:



```
;; zeile 4
;; zeile 5
;; zeile 1
;; zeile 2
;; zeile 3
;; zeile 1
;; zeile 2
;; zeile 3
```

Hier sieht man, dass die obersten drei Zeilen nun am oberen Rand verschwunden sind und unter den zuvor kopierten Block verschoben wurden.

Nun will ich Ihnen noch zeigen, wie man einen Block löschen kann. Ich möchte dies anhand der obersten zwei Zeilen (zeile 4 und zeile 5) demonstrieren.

Dazu definieren wir diese beiden Zeilen, wie zuvor beschrieben, als Block und drücken dann die Pfeil-Taste gefolgt von der Taste B.

Es erscheint wieder die Abfrage, welche Aktion man ausführen möchte und dieses mal drücken wir die Taste K, da wir den Block löschen wollen.

Vor dem tatsächlichen Löschen erscheint noch eine Sicherheitsabfrage:



Diese bestätigen wir durch Drücken der Taste Y und nach dem Löschen des Blocks, sollte sich im Editor folgendes Bild zeigen:



```
;; zeile 1
;; zeile 2
;; zeile 3
;; zeile 1
;; zeile 2
;; zeile 3
```

Wie Sie sehen können, sind die beiden Zeilen mit den Inhalten zeile 4 und zeile 5 verschwunden.

8.10 Schnell-Navigation innerhalb des Assembler-Programms

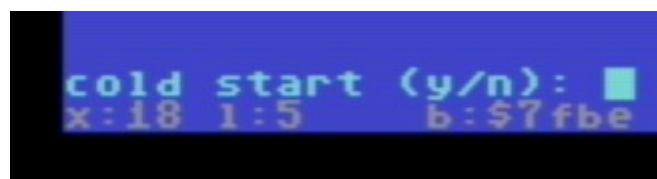
| Funktionstaste | Wirkung |
|----------------|--|
| F1 | Bewegt den Cursor um 20 Zeilen nach oben |
| F2 | Bewegt den Cursor zum Beginn des Programms |
| F3 | Bewegt den Cursor um 200 Zeilen nach oben |
| F5 | Bewegt den Cursor um 200 Zeilen nach unten |
| F7 | Bewegt den Cursor um 20 Zeilen nach unten |
| F8 | Bewegt den Cursor zum Ende des Programms |

8.11 Einen Kaltstart des Turbo Macro Pro durchführen

Durch Drücken der Pfeil-Taste gefolgt von der Taste C können Sie einen Kaltstart des Turbo Macro Pro durchführen, d.h. er wird auf jenen Zustand zurückgesetzt, in dem er sich direkt nach dem Laden von der Diskette befand.

Das aktuell im Editor befindliche Assembler-Programm wird dadurch natürlich gelöscht.

Wegen der drastischen Auswirkungen wird nach dem Drücken der Tastenkombination noch eine Sicherheitsabfrage angezeigt, ob man den Kaltstart wirklich durchführen will.



Nun beherrschen Sie die wichtigsten Funktionen, um mit dem Turbo Macro Pro erfolgreich Assembler-Programme erstellen bzw. ausführen zu können und sie werden sehen, dass das Programmieren im Vergleich zum SMON, vor allem bei größeren Programmen, viel mehr Spaß macht.

9 Sprites

Nach soviel grauer Theorie kommt nun wieder Bewegung ins Spiel!

9.1 Was sind Sprites?

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

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

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

9.2 Einfarbige Sprites

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

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

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

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

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

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

In vertikaler Richtung beträgt die Auflösung 21 Pixel, d.h. wir benötigen insgesamt (3 x 21 Bytes = 63 Bytes) um das Aussehen unseres Sprites festzulegen.

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

Unser Sprite-Raster sieht folgendermaßen aus:

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

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

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

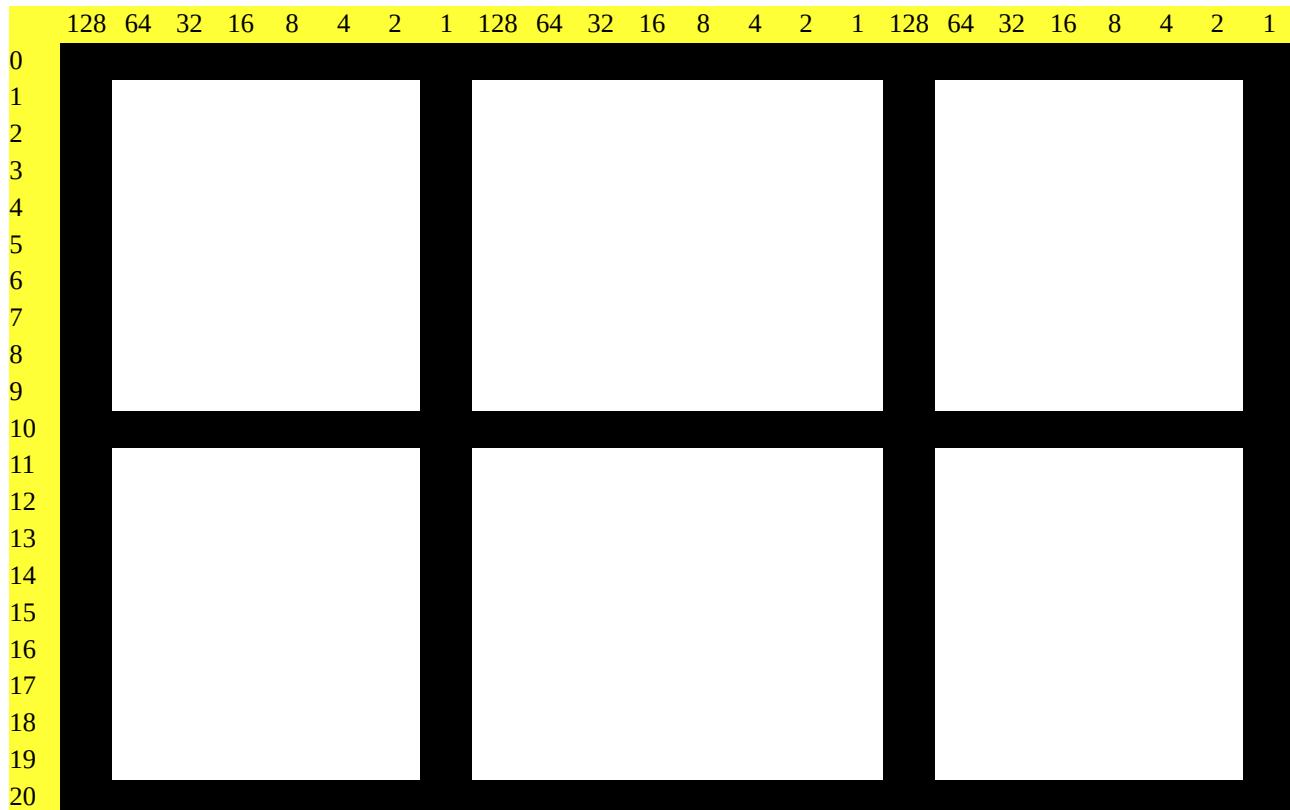
Diese beginnt jeweils mit 128 (2 hoch 7) und endet jeweils mit 1 (2 hoch 0)

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

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

9.3 Erstellung von Sprites

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



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

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

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

Gehen wir nun zum ersten Byte in der zweiten Zeile.

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

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

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

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

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

| Erstes Byte | Zweites Byte | Drittes Byte |
|--------------------|---------------------|---------------------|
| 255 | 255 | 255 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 129 | 0 | 129 |
| 255 | 255 | 255 |

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

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

Hinweis:

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

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

9.4 Auswahl des Speicherblocks für die Spritedaten

Nun müssen wir diese Daten an einem passenden Platz im Speicher ablegen.

Aber wo? Und wie sagen wir dann dem C64 wo wir die Daten für unser Sprite abgelegt haben?

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

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

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

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

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

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

Doch alles schön der Reihe nach.

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

Der Grund ist folgender:

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

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

Jede dieser Speicherstellen enthält eine Blocknummer zwischen 0 und 255.

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

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

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

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

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

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

| Angewandte (verwendete) | | | |
|-------------------------|----------------|--|---|
| \$2C0 - \$2FF | 704 - 767 | | Platz für Spritedatenblock 11, da nicht genutzt |
| 0000 0000 0000 | 0000 0000 0000 | | |

Um es gleich vorweg zu nehmen:

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

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

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

Festlegen der Blocknummer für die Spritedaten

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

Wir fügen also folgende Zeile hinzu:

```
10 POKE 2040,11
```

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

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

Das Kopieren führen wir mittels folgender Schleife durch:

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

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

9.5 Typ des Sprites festlegen (einfarbig oder mehrfarbig)

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

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

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

Dazu brauchen wir folgende UND-Verknüpfung:

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

9.6 Farbe des Sprites festlegen

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

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

70 POKE 53287,1

9.7 Festlegen der Spriteposition

Die Position eines Sprites wird durch eine Pixelposition in horizontaler und durch eine Pixelposition in vertikaler Richtung angegeben. In horizontaler Richtung (X) sind Werte von 0 bis 511 möglich und in vertikaler Richtung (Y) sind es Werte zwischen 0 und 255, wobei die Position X=0, Y=0 in der linken oberen Ecke des Bildschirms liegt.

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

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

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

Die Einstellung der X-Koordinate führen wir mit dem Befehl

80 POKE 53248,24

durch.

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

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

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

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

90 POKE 53249,50

durch.

9.8 Festlegen der Sprite-Priorität in Bezug auf den Hintergrund

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

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

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

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

9.9 Aktivierung von Sprites

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

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

Wichtig:

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

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

110 POKE 53269, PEEK(53269) OR 1

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

```
1160 DATA 129,0,129
1170 DATA 129,0,129
1180 DATA 129,0,129
1190 DATA 129,0,129
1200 DATA 129,0,129
1210 DATA 255,255,255
READY.
10 POKE 2040,11
20 FOR I=0 TO 62
30 READ A
40 POKE 704+I,A
50 NEXT I
60 POKE 53276,PEEK(53276) AND (NOT(1))
70 POKE 53287,1
80 POKE 53248,24
90 POKE 53249,50
100 POKE 53275,PEEK(53275) AND (NOT(1))
110 POKE 53269,PEEK(53269) OR 1
RUN
READY.
```

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

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

POKE 53287,7

ein.

Das Sprite sollte nun in gelb angezeigt werden.

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

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

Das Sprite sollte nun verschwunden sein.

Sichtbar machen können wir es wieder mit dem Befehl

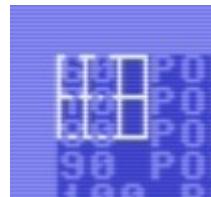
POKE 53269,PEEK(53269) OR 1

Das Sprite sollte nun wieder zu sehen sein.

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

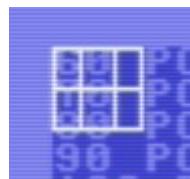
POKE 53275,PEEK(53275) OR 1

nötig.



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

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



Experimentieren wir nun ein wenig mit der Position des Sprites.

Verändern wir doch mal die X-Koordinate auf den Wert 100, was über den Befehl
POKE 53248,100

möglich ist.



Wichtig:

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

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

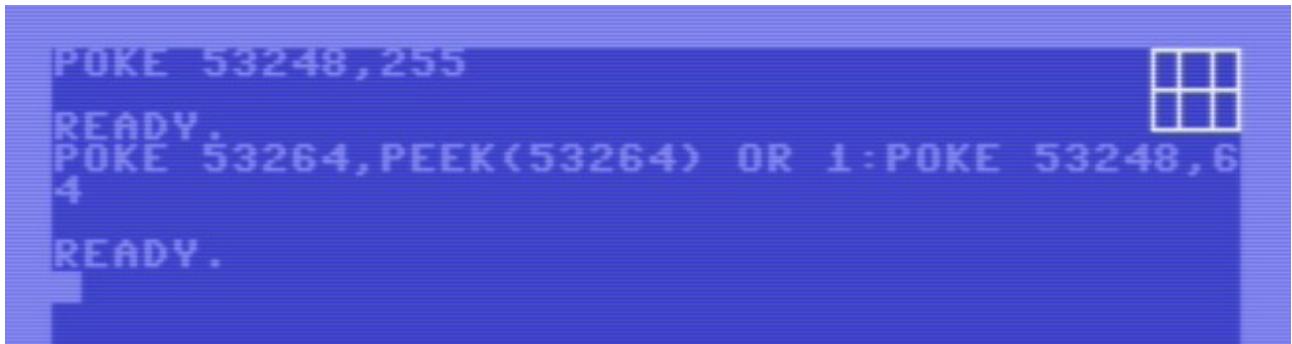


Wollen wir das Sprite an den rechten Rand des sichtbaren Bereichs positionieren, also auf die Position 320, dann müssen wir die Speicherstelle 53264 zu Hilfe nehmen.

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

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

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



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

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

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

```
|POKE 53248,255  
READY.  
POKE 53264,PEEK(53264) OR 1:POKE 53248,6  
4  
READY.  
POKE 53249,229  
READY.
```



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

9.10 Horizontale Vergrößerung von Sprites

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

Hier eine Vergrößerung des Sprites in horizontaler Richtung:

```
|POKE 53277,PEEK(53277) OR 1  
READY.
```



9.11 Vertikale Vergrößerung von Sprites

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

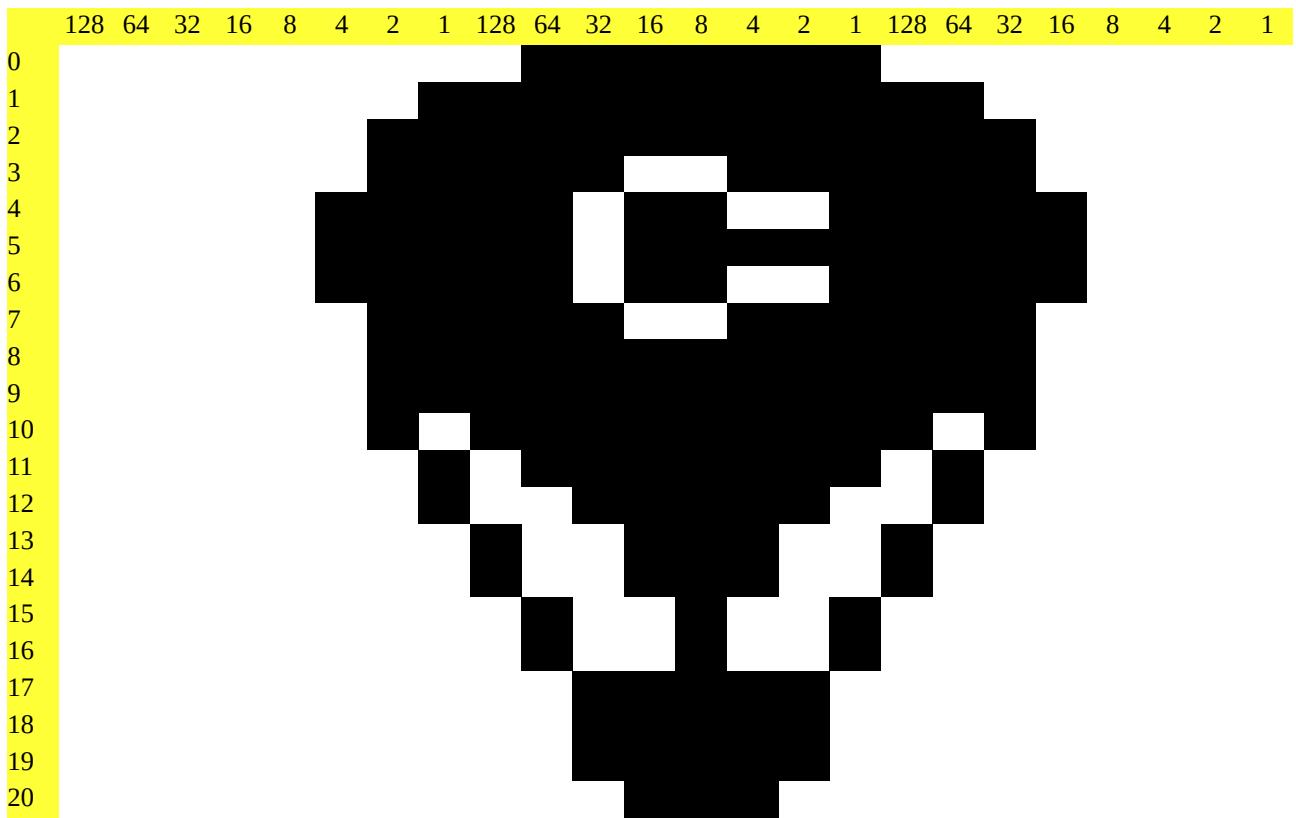
Hier eine Vergrößerung des Sprites in vertikaler Richtung:



9.12 Arbeiten mit mehreren Sprites

Fügen wir nun ein weiteres Sprite hinzu.

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



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

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

Hinweis:

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

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

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

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

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

Wir ergänzen also folgende Zeile:

120 POKE 2041,13

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

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

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

Typ des Sprites festlegen (einfarbig oder mehrfarbig)

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

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

Farbe des Sprites festlegen

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

180 POKE 53288,7

Festlegen der Spriteposition

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

190 POKE 53250,160

200 POKE 53251,140

Festlegen der Sprite-Priorität in Bezug auf den Hintergrund

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

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

Sprite aktivieren

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

220 POKE 53269,PEEK(53269) OR 2

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

```

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

```

READY.

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

Das macht die Ermittlung der Werte für die DATA-Zeilen natürlich auch recht einfach.
Wir benötigen für die 21 DATA-Zeilen immer denselben Inhalt 255,255,255.

Hinweis:

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

The screenshot shows a terminal window on a Commodore 64. The code displayed is an assembly program named SPRITE3BAS. It consists of several lines of assembly language, primarily using the DATA command to define memory blocks. The code is as follows:

```
1430 DATA 0,28,0
1440 REM DATEN FUER SPRITE 2
1450 DATA 255,255,255
1460 DATA 255,255,255
1470 DATA 255,255,255
1480 DATA 255,255,255
1490 DATA 255,255,255
1500 DATA 255,255,255
1510 DATA 255,255,255
1520 DATA 255,255,255
1530 DATA 255,255,255
1540 DATA 255,255,255
1550 DATA 255,255,255
1560 DATA 255,255,255
1570 DATA 255,255,255
1580 DATA 255,255,255
1590 DATA 255,255,255
1600 DATA 255,255,255
1610 DATA 255,255,255
1620 DATA 255,255,255
1630 DATA 255,255,255
1640 DATA 255,255,255
1650 DATA 255,255,255
READY.
```

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

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

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

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

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



9.13 Prioritäten von Sprites

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



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

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

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

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

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

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

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

9.14 Beispiel-Programm (einfarbige Sprites)

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

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

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

Folgende Schritte werden im Programm durchlaufen:

Vergeben der Blocknummern für die drei Sprites

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

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

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

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

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

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

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

Umkopieren der Spritedaten

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

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

```

    ldx #$00
copyloop lda SPRitegitter,x
            sta $02c0,x

            lda SPRiteballon,x
            sta $0340,x

            lda SPRiteviereck,x
            sta $0380,x

            inx
            cpx #$3f
            bne copyloop

```

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

Sprites einschalten

```

; sprite 0,1,2 einschalten

lda $d015
ora #$07
sta $d015

```

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

Sprites als einfarbig definieren

```

; alle sprites einfarbig

lda $d01c
and $f8
sta $d01c

```

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

Farben für die Sprites vergeben

```

; farbe fuer gitter

lda #$01
sta $d027

; farbe fuer ballon

lda #$07
sta $d028

; farbe fuer viereck

lda #$03

```

```
sta $d029
```

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

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

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

Priorität der Sprites gegenüber dem Hintergrund einstellen

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

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

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

Positionieren der Sprites

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

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

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

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

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

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

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

Dieses Schema gilt analog auch für die anderen Sprites.

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

Unterprogramm setbit8forx

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

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

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

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

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

```

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

```

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

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

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

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

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

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

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

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

Unterprogramm setspritex

```

; unterprogramm setspritex
; setzt die x-koordinate fuer ein sprite

; parameter:
; akku = spritenummer 0..7
; speicherstelle $fa = lo(x)
; speicherstelle $fb = hi(x)

setspritex
    pha
    asl a
    tay
    lda $fa
    sta $d000,y
    lda $fb
    beq xk19255
    pla
    sec

```

```

        jsr setbit8forx
        jmp setxende
xk1g255    pla
            cld
            jsr setbit8forx
setxende    rts

```

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

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

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

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

Dies ist der neue Befehl ASL (Arithmetic Shift Left)

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

Hier ein Beispiel:

Angenommen der Akkumulator enthält den binären Wert %10110010:

| | | | | | | | | |
|-----------------|---|---|---|---|---|---|---|---|
| Vor ASL | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| Nach ASL | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |

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

Auf der rechten Seite kam ein 0-Bit herein.

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

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

| | | | | | | | | |
|-----------------|---|---|---|---|---|---|---|---|
| Vor LSR | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| Nach LSR | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

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

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

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

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

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

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

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

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

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

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

Enthält das höherwertige Byte jedoch einen Wert ungleich 0, handelt es sich also um eine x-Koordinate, die größer als 255 ist, dann wird ebenfalls zunächst die zuvor auf dem Stack gesicherte Spritenummer mit dem Befehl PLA in den Akkumulator geholt, da wir sie für den Aufruf des

Unterprogramms setbit8forx brauchen. Da es sich um eine x-Koordinate größer als 255 handelt, muss das dem Sprite zugehörige Bit in der Speicherstelle \$D010 gesetzt werden.

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

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

Unterprogramm setspritey

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

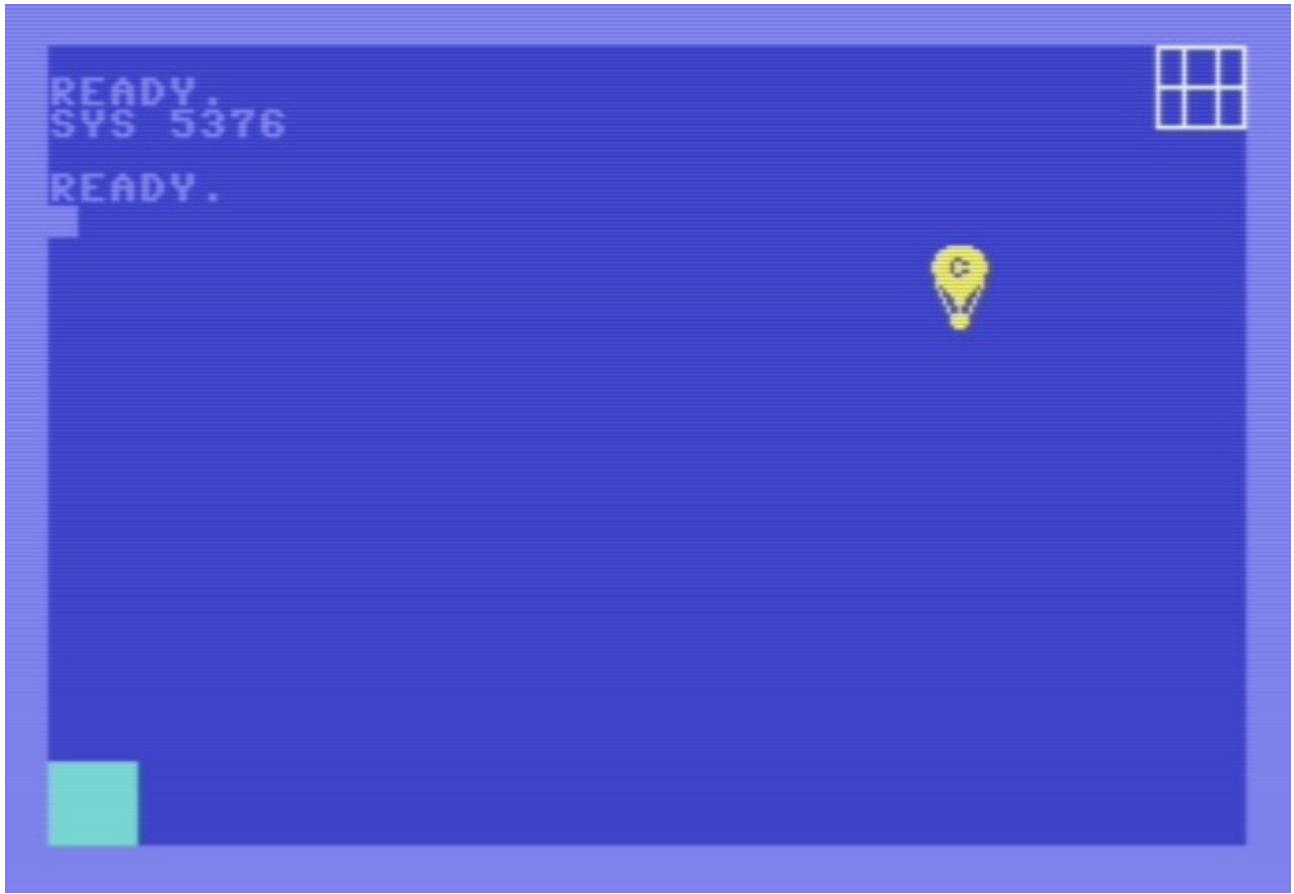
Dann schreiben wir die y-Koordinate in die Speicherstelle \$FA, laden den Akkumulator mit der Spritenummer 0 und rufen das Unterprogramm setspritey auf.

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

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

Den Abschluss des Programms bildet der Befehl RTS.

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



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

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

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

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

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

Unterprogramm incxpos

```
; unterprogramm incxkoord
; "increase xkoord"
; erhöht die x-koordinate des sprites
; um 1, die aktuelle x-koordinate ist
; in der variablen xpos gespeichert

incxkoord
    clc
    lda xkoord
    adc #$01
    sta xkoord
    bcc incxkoordende
    clc
    lda xkoord+1
    adc #$01
    sta xkoord+1

incxkoordende
    rts
```

Am Ende des Codes ist folgende Variable definiert:

```
; x-koordinate des sprites
; wird bei der horizontalen
; bewegung laufend erhöht

xkoord
    .byte $18,$00
```

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

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

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

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

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

Ergebnis wird sogleich mit dem Befehl STA xkoord in das niedrige Byte der Variablen xkoord zurückgeschrieben.

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

1.) Der Akkumatorinhalt ist kleiner als 255, d.h. es tritt bei der Erhöhung um 1 kein Übertrag auf (das Carryflag wird also nicht gesetzt)

2.) Der Akkumatorinhalt ist gleich 255, d.h. er springt durch die Erhöhung um 1 auf 0 zurück und das Carryflag wird gesetzt, um den Übertrag anzuzeigen.

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

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

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

Unterprogramm rbreached

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

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

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

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

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

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

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

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

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

Unterprogramm delay

```
; unterprogramm delay
;

delay      ldy #$00

loopwait   iny
            cpy #$ff
            bne loopwait
            rts
```

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

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

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

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

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

lda #$00
jsr setspriteX

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

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

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

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

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

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

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

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

```
; vor start auf taste
; warten

loopwaitkey
    jsr $ffe4
    beq loopwaitkey
```

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

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

```
; sprite horizontal bewegen

loophoriz
; x-koordinate um 1 erhöhen
    jsr incxkoord
; sprite auf die neue
; x-koordinate setzen
    lda xkoord
    sta $fa ; lo(x) in $fa
    lda xkoord+1
    sta $fb ; hi(x) in $fb
    lda #$00 ; spritenr im akku
    jsr setspritex
; kurz warten bis zum
; nächsten bewegungsschritt
    jsr delay
; hat das sprite
; den rechten bildschirmrand
; erreicht?
```

```
jsr rbreached
    cpy #\$01
    ; wenn nicht dann naechster
    ; bewegungsschritt nach rechts
    bne loophoriz
rts
```

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

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

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

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

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

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

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

9.15 Mehrfarbige Sprites

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

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

Hier ein Beispielraster für ein mehrfarbiges Sprite:

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

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

Enthält eine Zelle die Bitkombination 10, dann enthält der Pixel an dieser Stelle jene Farbe, welche in dem Farbregister, welches dem Sprite zugeordnet ist, hinterlegt ist.

Dies sind dieselben Register wie bei den einfärbigen Sprites, d.h. Register 53287 enthält die Farbinformation für Sprite 0, Register 53288 enthält die Farbinformation für Sprite 1 usw.bis hin zur Register 53294, welches die Farbinformation für Sprite 7 enthält. In dem Beispiel sind das die grünen Zellen.

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

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

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

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

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

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

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

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

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

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

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

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

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

Hinweis:

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

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

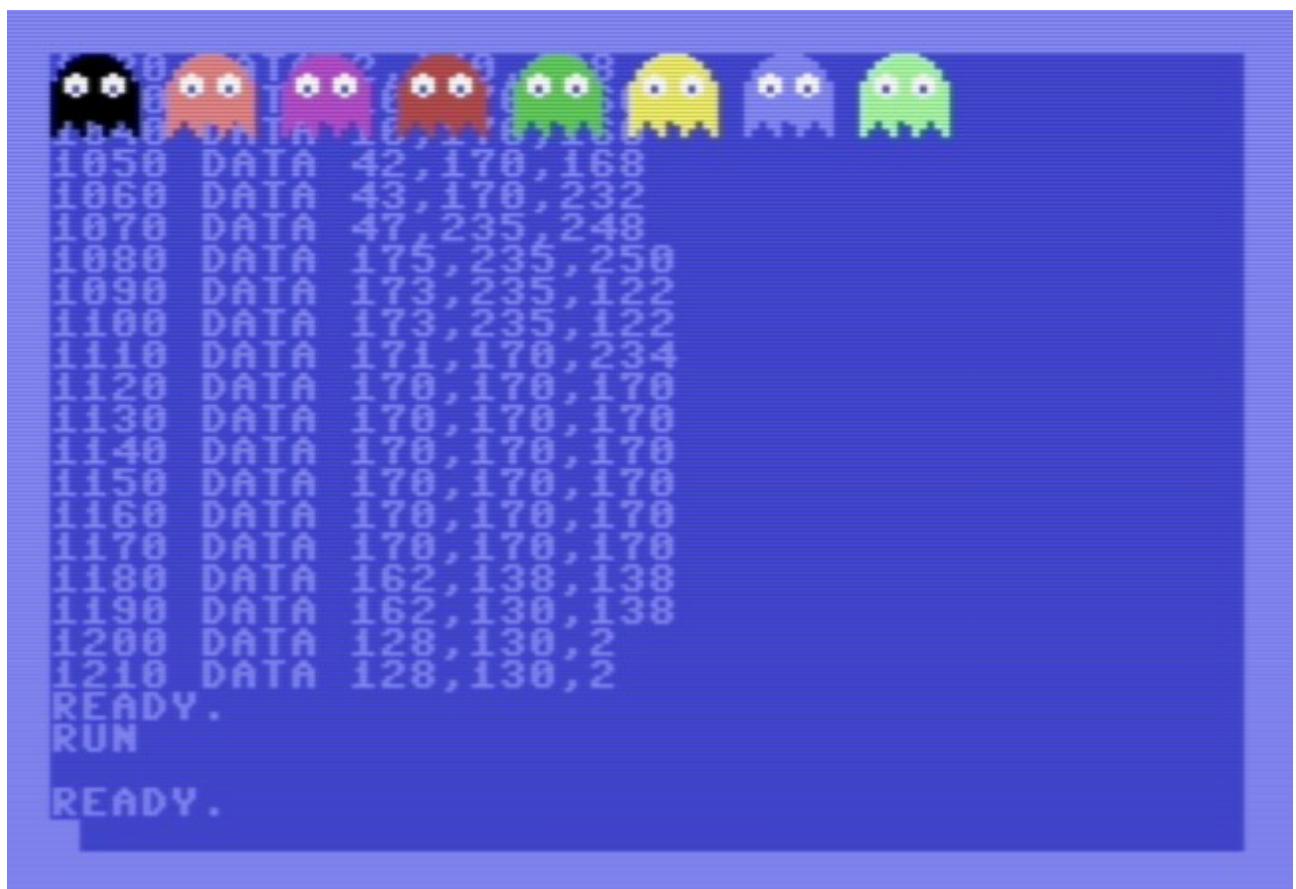
9.16 Beispiel-Programm (mehrfarbige Sprites)

Was soll unser Programm machen?

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

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

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



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

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

Zeile 10 – 40

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

Zeile 50 – 90

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

Zeile 100 – 110

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

Zeile 120 – 130

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

Zeile 140 – 150

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

Zeile 160 – 170

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

Zeile 180 – 330

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



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

READY.
```

Zeile 340 – 490

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


```
|LIST 340-510
```

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

```
READY.
```

Zeile 500 – 510

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

Zeile 520 – 790

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

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

```

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

```

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

```

510 PUKE 53269,255
515 GET A$:IF A$="" THEN 515
520 REM SPRITES NACH UNTEN BEWEGEN

```

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

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

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

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

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

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

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

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

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

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

Kommen wir nun zum Assembler-Code.

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

Wir verwenden gleich den Block 11.

```

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

idx #$00
lda #$0b

loopblocknr
    sta $07f8,x
    inx
    cpx #$08
    bne loopblocknr

```

Anschließend kopieren wir die Spritedaten in diesen Block.

```

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

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

```

Dann aktivieren wir alle Sprites.

```

; alle sprites einschalten

lda #$ff
sta $d015

```

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

```

; alle sprites sind mehrfarbig

lda #$ff
sta $d01c

```

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

```

; individuelle spritefarben
; setzen

idx #$00

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

```

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

```
; gemeinsame farbe 1 setzen  
lda gemfarbei  
sta $d025  
  
; gemeinsame farbe 2 setzen  
lda gemfarbe2  
sta $d026
```

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

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

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

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

```
; sprites an die  
; ausgangspositionen  
; setzen  
  
; x-koordinaten der sprites  
; setzen  
  
ldx #$10  
stx $d000 ; x fuer sprite 0  
  
ldx #$37  
stx $d002 ; x fuer sprite 1  
  
ldx #$56  
stx $d004 ; x fuer sprite 2  
  
ldx #$75  
stx $d006 ; x fuer sprite 3  
  
ldx #$94  
stx $d008 ; x fuer sprite 4  
  
ldx #$b3  
stx $d00a ; x fuer sprite 5  
  
ldx #$d2  
stx $d00c ; x fuer sprite 6  
  
ldx #$f1  
stx $d00e ; x fuer sprite 7
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Anschließend wird geprüft, ob die Sprites bereits den unteren Bildschirmrand erreicht haben. Dies ist dann der Fall, wenn im Y Register der Wert \$E5 steht.

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

```

    cpy #$e5
    ; wenn nicht dann naechster
    ; bewegungsschritt nach unten
    bne loopdown

```

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

```

; sprites wieder zum oberen
; bildschirmrand bewegen

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

```

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

10 Ein Sprite-Editor als Abschlussprojekt

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

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



Am besten ist es jedoch, wenn Sie das Programm von der Diskette namens TMP mit LOAD „SPRITEEDITOR“,8,1 laden, mit SYS 12000 starten und einfach mal ausprobieren. Auf diese Art und Weise lernen Sie das Programm am besten kennen. 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 :)

Nachdem Sie den Spriteeditor geladen und gestartet haben, legen Sie bitte die Diskette namens SPRITEEDITOR ein.

Hier eine kurze Bedienungsanleitung des Spriteeditors:

- **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 12000 aufrufen.

Erstellen Sie nun ein einfaches Sprite und speichern es mit der Tastenkombination SHIFT + S. Für die Dauer des Speichervorgangs wird der Bildschirmrahmen in einer anderen Farbe dargestellt und sobald der Speichervorgang abgeschlossen ist, wechselt die Farbe wieder zurück auf die vorherige Farbe. Die Spritedaten wurden nun in einer Datei namens SPRITE gespeichert.

Vorab eine Anmerkung in Bezug auf den Begriff „Cursor“

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

Anmerkung: Alle ab jetzt genannten Assembler- und Basic-Programme befinden sich auf der Diskette namens SPRITEEDITOR.

Auf der Diskette befindet sich 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. Laden Sie also das Programm namens SHOWSPRITE und nachdem Sie es gestartet haben, sollten Sie das Sprite, welches Sie soeben erstellt haben, auf dem Bildschirm zu sehen bekommen.

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

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

Beginnen wir also mit dem ersten Teil.

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

Für die ersten Schritte erstellen wir folgende Unterprogramme:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$18 * 32 = 576$$

$$18 * 8 = 144$$

$$576 + 144 = 720$$

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

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

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

Inhalt des Akkumulators vor und nach Ausführung des Befehls ASL:

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

Inhalt des Carryflags nach der Ausführung: 1

Was passiert nun beim Ausführen des Befehls ASL?

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

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

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

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

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

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

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

Doch wie führen wir diese Verschiebung durch? Ein erster Gedanke wäre, das höherwertige Byte und das niedlerwertige 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 niedlerwertige Byte %1100 0100 um eine Bitposition nach links schieben, dann ergeben sich folgende Werte:

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

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

Wenn wir das höherwertige Byte %0000 1001 um eine Bitposition nach links schieben, dann ergeben sich folgende Werte:

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

Auf der rechten Seite ist ein Bit mit dem Inhalt 0 hereingewandert (orange markiert) und das grün markierte Bit fällt auf der linken Seite heraus und wandert ins Carryflag. In diesem Fall hat dieses nun den Inhalt 0, weil das grün markierte 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 %1100 0100 durch den Befehl ASL 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 dieses Bit in die ganz rechts liegende Position des höherwertigen Bytes wandern müssen.

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

Doch wie lösen wir das Problem nun?

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

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

Hier zur Veranschaulichung ein Beispiel, bei dem wir wieder den Wert 200 (binär %1100 1000 bzw. hexadezimal \$C8) im Akkumulator verwenden wollen.

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

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

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

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

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

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

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

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

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

Doch zurück zur vorherigen Aufgabe, bei der wir die Bits der dezimalen Zahl 2500 um eine Stelle nach links verschieben wollten. 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 bei der Verschiebung des niedlerwertigen Bytes verlorengegangen ist, steht nach der Ausführung des Befehls ja trotzdem 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 niedlerwertige Byte den Inhalt 1 und der Inhalt des höherwertigen Bytes vor Anwendung des Befehl ROL lautet %0000 1001.

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

| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----------|---|---|---|---|---|---|---|---|
| vor ROL | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| nach ROL | 0 | 0 | 0 | 1 | 0 | 0 | 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  
; shiftet eine 16bit zahl um eine  
; bestimmte anzahl an stellen  
; nach links  
  
; parameter:  
; zahl: lo/hi in $fd/$fe  
; stellen: x register  
  
; ruckgabewerte:  
; geshiftete zahl: lo/hi in $fd/$fe  
  
; aendert:  
; x,status  
  
as16  
as16_loop  
    ; lo byte shiften  
    asl $fd  
    ; hi byte shiften  
    rol $fe  
    dex  
    bne as16_loop  
    rts
```

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

Das Unterprogramm bietet die Möglichkeit, eine 16bit Zahl nicht nur um eine, sondern um mehrere Stellen zu verschieben. Die Anzahl der Stellen übergeben wir als Parameter im X Register.

Wir bilden also eine Schleife, die bei jedem Durchlauf die Bits der 16bit Zahl um eine weitere Position nach links verschiebt.

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

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

Um nochmal auf die Bitrotation zurückzukommen:

So wie es für den Befehl ASL ein Gegenstück in Form des Befehls LSR gibt, existiert auch für den Befehl ROL ein Gegenstück namens ROR, welcher eine Bitrotation in die andere Richtung, also nach rechts, bewirkt. Bei der Ausführung fällt nicht wie beim Befehl ROL das Bit 7 auf der linken Seite heraus, sondern das Bit 0 auf der rechten Seite und wandert in das Carryflag. Mit seiner Hilfe könnten sie eine 16bit Version des Befehls LSR bauen. Damit wären dann Divisionen durch Zweierpotenzen möglich.

Sowohl der Befehl LSR als auch der Befehl ROR kommen beim Sprite-Editor nicht zur Anwendung, es wäre jedoch eventuell eine gute Übung für Sie, die genannte 16bit Version des Befehls LSR umzusetzen.

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.

Der Befehl ADC lässt den Inhalt des Carryflags in die Addition miteinfließen und deswegen ist es vor der Durchführung einer 8bit Addition wichtig, das Carryflag zurückzusetzen. Dies hängt jedoch vom Anwendungsfall ab.

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. Wäre das Carryflag hingegen aus irgendeinem Grund 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 |

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.

Vor der Addition der beiden höherwertigen Bytes darf das Carryflag also nicht zurückgesetzt werden, da es ja bei der Addition der beiden niederwertigen Bytes zu einem Überlauf gekommen sein kann.

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  
;  
; rückgabewerte:  
; summe: lo/hi in $fb/$fc  
;  
; ändert:  
; 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 hinzugefügt. 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 hinzugefügt. Falls es bei der Addition der niederwertigen Bytes zu einem Überlauf kam, das Carryflag also gesetzt wurde, fließt dieses in die Addition mit ein.

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

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

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

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

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

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

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

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

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

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

Als nächstes addieren wir zu diesem Wert jenen Wert, den wir als Parameter für die Spalte übergeben haben und zuletzt müssen wir noch die Startadresse des Bildschirmspeichers, welche im Normalfall der Adresse 1024 entspricht, hinzuaddieren. Dann haben wir endlich die gewünschte Adresse im Bildschirmspeicher.

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

Die Adresse ließe sich dann mit der Formel

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

berechnen.

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

```
;-----  
; calcposaddr  
; berechnet die adresse einer  
; Position zeile/spalte im  
; bildschirmspeicher  
  
; Parameter:  
; spalte: x register  
; zeile: y register  
  
; Rueckgabewerte:  
; adresse: lo/hi im x/y register  
  
; aendert:  
; a,x,y,status,$fb,$fc  
  
calcposaddr  
    ; speicherstellen $fb und $fc  
    ; auf dem stack sichern, da  
    ; sie in diesem unterprogramm  
    ; ueberschrieben werden  
  
    lda $fb  
    pha
```

```

lda $fc
pha

; auch das x register und das
; y register (welche die
; Parameter fuer zeile und
; spalte beinhalten) auf dem
; stack sichern, weil sie
; durch die nachfolgenden
; berechnungen ueberschrieben
; werden

txa
pha

tya
pha

; zeile * 32 berechnen
; lo (zeile) nach $fd
sty $fd

; hi (zeile) nach $fe
; ist immer $00 weil der
; wert fuer die zeile nur von
; 0 bis 24 reicht

ldy #$00
sty $fe

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

idx #$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

idx #$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 adci6

; 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 adci6

; 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 adci6

; 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

```

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) hinzufügen. 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 programs  
    ; 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:

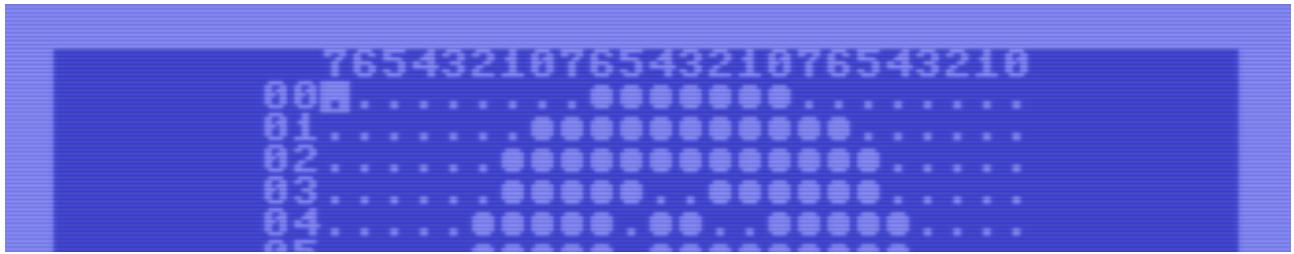


Wie wir bereits wissen, bestehen die Spritedaten aus 21 Reihen zu je 3 Bytes. Der String am oberen Rand stellt die Bitpositionen innerhalb dieser drei Bytes dar.

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.

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



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

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

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

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

Dieses möchte ich Ihnen nun ohne Umschweife vorstellen.

```
;-----  
; printstr  
; gibt einen null-terminierten string  
; an der aktuellen cursorposition  
; aus  
  
; parameter:  
; adresse des strings: lo/hi in $fd/fe  
; spalte: y register  
; zeile: x register  
  
; rückgabewerte:  
; keine  
  
; ändert:  
; 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. In den folgenden Ausführungen meine ich wirklich den C64 Cursor, denn durch seine Position wird bestimmt, wo die nächste Ausgabe auf dem Bildschirm stattfindet.

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

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

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

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

Nun können wir uns an die Ausgabe des Strings machen. Über die 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.

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

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

Über diese Art der Adressierung haben wir die Möglichkeit, auf eine Speicherstelle zuzugreifen, deren Adresse in zwei aufeinanderfolgenden Speicherstellen innerhalb der Zeropage zu finden ist.

Das niedwertige 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 niedwertige 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 niedwertige Byte der Adresse hinterlegt haben.

Man kann sich das folgendermaßen vorstellen:

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

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

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

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

Angewandt auf das obige Beispiel würde das erste Postfach die Nummer \$FD und das zweite Postfach die Nummer \$FE haben. Auf dem Zettel aus dem ersten Postfach steht das niederwertige Byte der Adresse (Postleitzahl und Ort) und auf dem Zettel aus dem zweiten Postfach steht das höherwertige Byte (Straße und Hausnummer).

Zusammengesetzt ergeben diese Informationen dann die gewünschte Adresse.

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

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

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

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

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

```

; niedwertiges byte der
; Adresse in die
; Speicherstelle $fb schreiben
lda #$01
sta $fb

; hoherwertiges 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 niedwertige Byte der Adresse (\$01) in die Speicherstelle \$FB und das höherwertige Byte der Adresse (\$04) in die Speicherstelle \$FC.

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

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

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

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

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

Die CPU bildet die Adresse dann durch die Formel

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

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

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

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

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

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

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

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
idx #$00
idy #$00
jsr $ffff0

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

; ruckgabewerte:
; keine

; aendert:
; a,y,status

printstr      ; cursor positionieren
              clc
              jsr $ffff0
              ; 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:

The screenshot shows the SMON assembly debugger interface. At the top, there is a memory dump section with addresses 1521 to 1523. Below it is an assembly code listing for address 1523, which contains a loop structure. The assembly code includes instructions like LDX #00, LDA 1500,X, STA 3000,X, INX, CPX #23, BNE 1525, and RTS. The code is labeled with A and F. At the bottom left, there is a small icon.

| | | | | |
|-------|------|----|-----|--------|
| ,1521 | D0 | F0 | BNE | 1513 |
| ,1523 | 60 | | RTS | |
| ----- | | | | |
| .A | 1523 | | | |
| 1523 | A2 | 00 | LDX | #00 |
| 1525 | BD | 00 | LDA | 1500,X |
| 1528 | 9D | 00 | STA | 3000,X |
| 152B | E8 | | INX | |
| 152C | E0 | 23 | CPX | #23 |
| 152E | D0 | F5 | BNE | 1525 |
| 1530 | A9 | 60 | LDA | #60 |
| 1532 | 8D | 23 | STA | 3023 |
| 1535 | 60 | | RTS | |
| ----- | | | | |
| .F | | | | |
| 1523 | A2 | 00 | LDX | #00 |
| 1525 | BD | 00 | LDA | 1500,X |
| 1528 | 9D | 00 | STA | 3000,X |
| 152B | E8 | | INX | |
| 152C | E0 | 23 | CPX | #23 |
| 152E | D0 | F5 | BNE | 1525 |
| 1530 | A9 | 60 | LDA | #60 |
| 1532 | 8D | 23 | STA | 3023 |
| 1535 | 60 | | RTS | |
| ----- | | | | |

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

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

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

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

Doch zurück zu den Zeichenfolgen #< und #>.

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

l da #<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 Kernel-Funktion CHROUT, deren Aufruf durch den Befehl JSR \$FFD2 erfolgt, der String Zeichen für Zeichen ausgegeben wird. Die Funktion CHROUT erwartet den PETSCII-Code des auszugebenden Zeichens im Akkumulator und gibt das entsprechende Zeichen dann an der aktuellen Position des C64 Cursors aus.

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

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

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 Kernel-Funktion \$FFD2 das Zeichen ausgegeben. Im Anschluss wird der Index im Y Register um 1 erhöht und dann zum Label printstr_loop gesprungen.

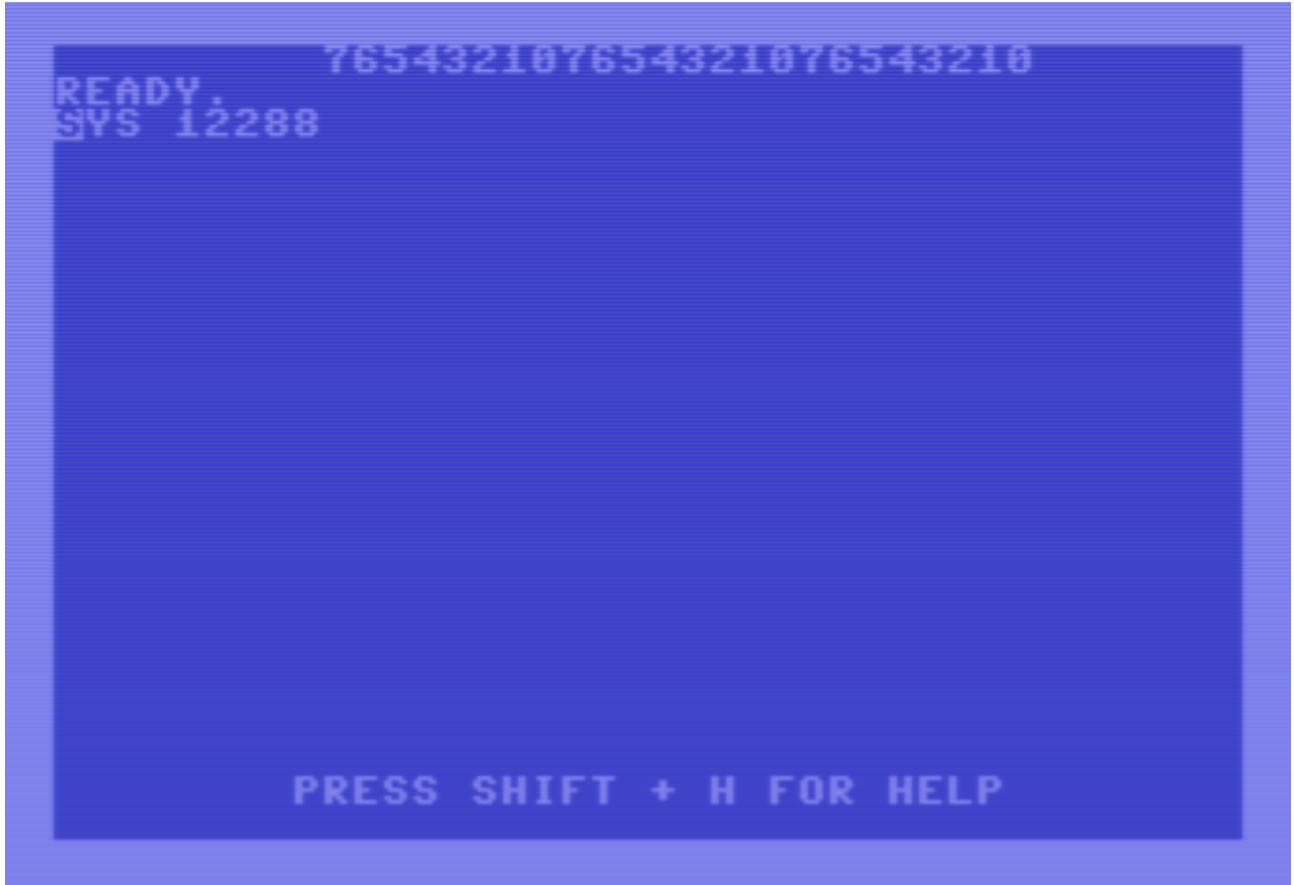
Nun ergibt sich für den nächsten Schleifendurchlauf die Speicheradresse

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 editorowstr 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 durchlufe 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.

```
editorowstr
    .text    " "
    .text    "0"
    .text    "1"
    .text    "2"
    .text    "3"
    .text    "4"
    .text    "5"
    .text    "6"
    .text    "7"
    .text    "8"
    .text    "9"
    .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 editorowstr 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 editorowstr kopiert werden.

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

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

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

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

```
editorow
    .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 editorow 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

idx scrrrow
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 scrrrow
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 editorowstr 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 editorowstr eingesetzt wird.

| Reihennummer | Index in Datenfeld rownrstr | Nummernstring aus Datenfeld rownrstr | editorowstr |
|--------------|--------------------------------|---|-------------|
| 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..... |

| Reihennummer | Index in Datenfeld rownrstr | Nummernstring aus Datenfeld rownrstr | editorrowstr |
|---------------------|--|---|---------------------|
| 7 | 14 | 07 | 07..... |
| 8 | 16 | 08 | 08..... |
| 9 | 18 | 09 | 09..... |
| 10 | 20 | 10 | 10..... |
| 11 | 22 | 11 | 11..... |
| 12 | 24 | 12 | 12..... |
| 13 | 26 | 13 | 13..... |
| 14 | 28 | 14 | 14..... |
| 15 | 30 | 15 | 15..... |
| 16 | 32 | 16 | 16..... |
| 17 | 34 | 17 | 17..... |
| 18 | 36 | 18 | 18..... |
| 19 | 38 | 19 | 19..... |
| 20 | 40 | 20 | 20..... |

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

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

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 editorowstr+1 an die zweite Stelle im String editorowstr.

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

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

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

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

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

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

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

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

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

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

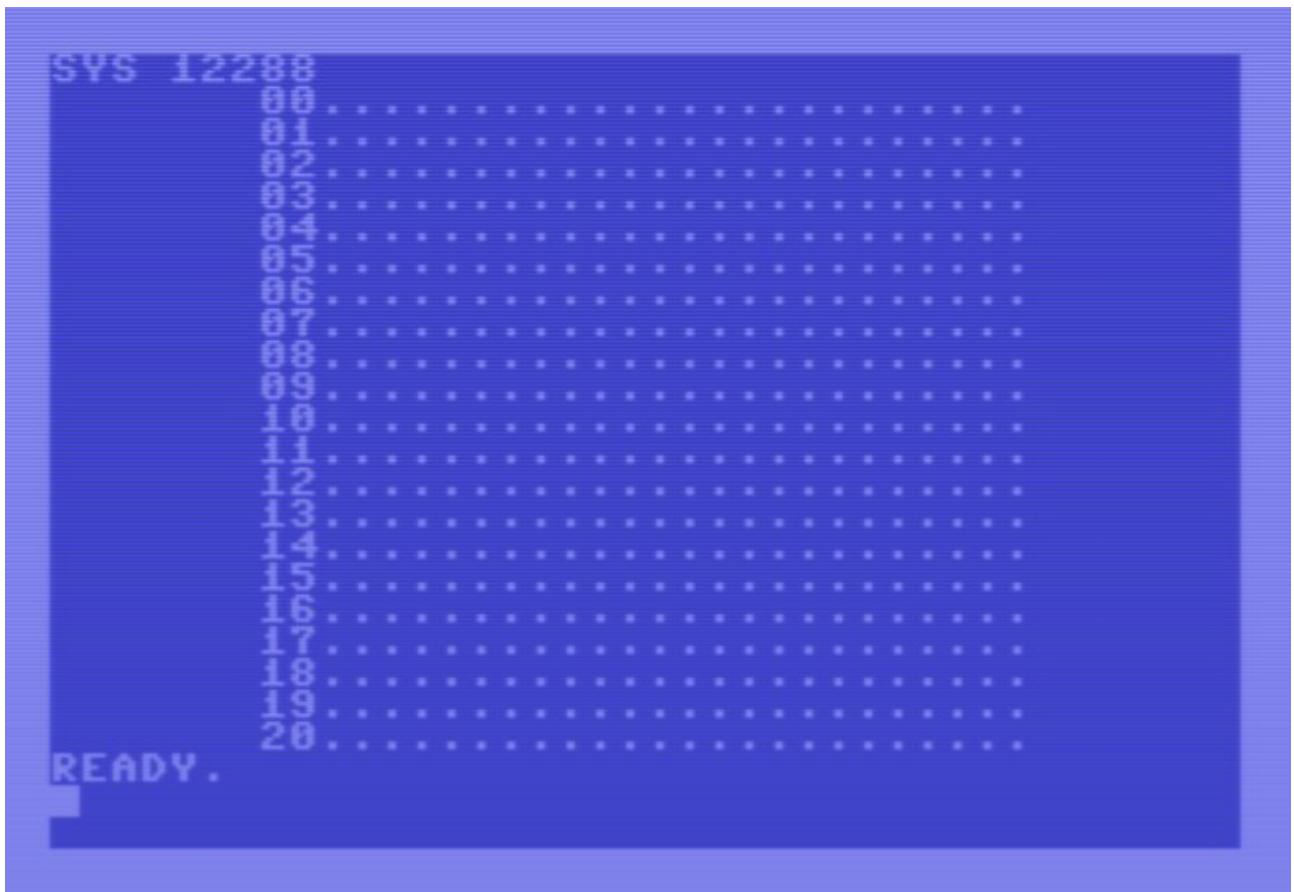
Falls jedoch noch nicht alle Reihen ausgegeben wurden, erhöhen wir die Reihennummer im Akkumulator um 1, um zur nächsten Reihe überzugehen.

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  
        idx #$00  
        idy #$00  
        jsr $ffff0  
  
        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  
    idx editorrow  
    idy 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

idx 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.

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

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

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

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

Bei jeder Cursorbewegung muss daher geprüft werden, ob durch diese die Grenzen des Editors eingehalten werden. Diese Grenzen sind jedoch abhängig von den Inhalten der Variablen editorrow und editorcol und deswegen werden wir ein Unterprogramm namens init schreiben, das beim Starten des Editors genau einmal aufgerufen wird und sämtliche Positionsangaben mit den korrekten Werten initialisiert.

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

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

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

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

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

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

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

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

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

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

Nachfolgend der Assembler-Code des Unterprogramms init:

```
;-----  
; init  
; berechnet die grenzen fuer die  
; cursorbewegung sowie die startadresse  
; des cursors im bildschirmspeicher,  
; die startposition des cursors wird  
; ebenfalls gesetzt  
  
; parameter:  
; keine  
  
; rueckgabewerte:  
; keine  
  
; aendert:  
; a,x,y,status  
  
init  
; grenzen fuer zeilenwert  
; des cursors berechnen und  
; zeilenwert fuer dessen  
; startposition setzen  
  
    lda editorrow  
    clc  
    adc #$01  
    sta csrminrow  
    sta csrrow  
    adc #$14  
    sta csrmaxrow  
  
; grenzen fuer spaltenwert
```

```

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

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

; adresse des cursors im
; bildschirmspeicher
; berechnen

idx csrcol
idy csrrow
jsr calcposaddr

; diese wird laufend bei
; jeder cursorbewegung
; in den speicherstellen
; $fb und $fc aktualisiert

stx $fb
sty $fc

; startposition des cursors
; merken

stx csrhomeaddr
sty csrhomeaddr+1

rts

```

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

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

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

Da nun in den Variablen csrrow und csrcol die Startposition des Cursors steht, können wir nun durch das Unterprogramm calcposaddr die Adresse im Bildschirmspeicher berechnen, welche der Startposition des Cursors entspricht.

Das niederwertige Byte steht laut Dokumentation, welche wir beim Unterprogramm calcposaddr hinterlegt haben, in der Speicherstelle \$FB und das höherwertige Byte in der Speicherstelle \$FC.

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

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

Doch bevor wir die Tastatur abfragen können, müssen wir zunächst die Tastaturcodes der gewünschten Tasten(kombinationen) ermitteln. Diese Codes können wie beispielsweise die Farbcodes in Tabellen nachgeschlagen werden.

Ich habe diejenigen, welche für den Sprite-Editor relevant sind, herausgesucht und in folgender Tabelle zusammengefasst:

| Funktion | Tasten(kombination) | Tastaturcode |
|-----------------------------|---------------------|--------------|
| Cursor nach rechts bewegen | Cursor nach rechts | \$1D |
| Cursor nach unten bewegen | Cursor nach unten | \$11 |
| Cursor nach links bewegen | Cursor nach links | \$9D |
| Cursor nach oben bewegen | Cursor nach oben | \$91 |
| Bit setzen | Return Taste | \$0D |
| Bit löschen | Leertaste | \$20 |
| Sprite in Datei speichern | SHIFT + Taste S | \$D3 |
| Sprite aus Datei laden | SHIFT + Taste L | \$CC |
| Editorbereich löschen | SHIFT + CLR/HOME | \$93 |
| Hilfsinformationen anzeigen | SHIFT + Taste H | \$C8 |
| Editor beenden | SHIFT + Taste Q | \$D1 |

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

```

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

```

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

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

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

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

```
quitkey
.byte $58
```

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

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

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

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

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

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

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

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

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

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

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

```
srcode_c1rscr
    .byte $93
```

Danach wird durch den Befehl RTS zum Aufrufer zurückgekehrt.

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

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

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

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

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

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

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

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

Subtraktion zweier 16bit Zahlen

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

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

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

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

Solch ein Unterlauf wird durch ein gelöschtes Carryflag signalisiert.

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

```
;-----  
; sbc16  
; subtrahiert zwei 16bit zahlen  
; zahl1-zahl2  
  
; parameter:  
; zahl1: lo/hi in $fb/$fc  
; zahl2: lo/hi im x/y register  
  
; rückgabewerte:  
; differenz: lo/hi in $fb/$fc  
  
; ändert:  
; a,status,$fd,$fe  
  
sbc16      stx $fd
```

```

sty $fe
; lo bytes subtrahieren
sec
lda $fb
sbc $fd
sta $fb
; hi bytes subtrahieren
lda $fc
sbc $fe
sta $fc
rts

```

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

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

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

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

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

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

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

Darstellung und Steuerung des Cursors

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

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

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

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

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

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

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

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

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

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

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

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

Nachfolgend der Assembler-Code des Unterprogramms showcsr:

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

Über die neue Adressierungsart, welche wir vor kurzem kennengelernt haben, wird hier der Inhalt aus jener Speicheradresse gelesen, an der der Cursor gerade steht. Diese Adresse ist, wie vorhin

erwähnt, in den Speicherstellen \$FB bzw. \$FC zu finden und durch den Befehl LDA (\$FB),Y wird der dortige Inhalt in den Akkumulator geladen.

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

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

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

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

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

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

```
jsr showcsr  
rts
```

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



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

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

Ich habe das Unterprogramm keyctrl ein wenig umgebaut:

```
keyctrl
keyloop      ; tastatur abfragen bis eine
              ; taste gedrueckt wird
    jsr $ffe4
    beq keyloop
              ; cursor nach rechts?
    cmp csrrightkey
    bne check_quit
    jsr csrright
    jmp keyloop

check_quit   ; editor beenden?
    cmp quitkey
    beq keyctrl_end
              ; ansonsten wieder mit der
              ; tastaturabfrage forsetzen
    jmp keyloop
```

```

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

lda scrcode_clrscr
jsr $ffd2

rts

```

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

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

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

Nachfolgend sehen Sie das Unterprogramm csrright:

```

-----
; csrright
; bewegt den cursor nach rechts
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
;
csrright
; pruefen ob cursor bereits
; am rechten rand des editors
; steht
    lda csrcol
    cmp csrMaxcol
;
; wenn ja, dann cursor lassen
; wo er ist und zurueck zum
; aufrufer
    beq csrright_end
;
; ansonsten cursor bewegen
;
; dazu erstmal an der
; aktuellen position entfernen
    jsr removecsr
;
; es geht nach rechts, also
; cursorspalte um 1 erhoehen
    inc csrcol
;
; nun auch die adresse im
; bildschirmspeicher
; aktualisieren und ebenfalls
; um 1 erhoehen

```

```

    ldx #$01
    ldy #$00
    jsr adc16
    ; cursor an neuer position
    ; anzeigen
    jsr showcsr
csrright_end
rts

```

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

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

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

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

Nun wird es interessant.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
;--  
; csrdown  
; bewegt den cursor nach unten  
;  
; parameter:  
; keine  
;  
; rueckgabewerte:  
; keine  
;  
; aendert:  
; a,x,y,status  
  
csrdown  
    lda csrrow  
    cmp CSRMaxRow  
    beq csrdown_end  
    jsr removeCSR  
    inc csrrow  
    ldx #$28  
    ldy #$00  
    jsr adci16  
    jsr showCSR  
  
csrdown_end  
    rts
```

```
;--  
; csrleft  
; bewegt den cursor nach links  
;  
; parameter:  
; keine  
;  
; rueckgabewerte:  
; keine  
;  
; aendert:  
; a,x,y,status  
  
csrleft  
    lda csrCol  
    cmp CSRMinCol  
    beq csrleft_end  
    jsr removeCSR  
    dec csrCol  
    ldx #$01
```

```

ldy #$00
jsr sbc16
jsr showcsr
csrleft_end
rts

```

```

;-----
; CSRUP
; bewegt den cursor nach oben
; parameter:
; keine
; rueckgabewerte:
; keine
; aendert:
; a,x,y,status
CSRUP
    lda csrrrow
    cmp CSRMINROW
    beq CSRUP_end
    jsr removeCSR
    dec csrrrow
    ldx #$28
    ldy #$00
    jsr sbc16
    jsr showCSR
CSRUP_end
rts

```

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

```

keyctrl
keyloop
    ; tastatur abfragen bis eine
    ; taste gedrückt wird
    jsr $ffe4
    beq keyloop

    ; cursor nach rechts?
    cmp csrrightkey
    bne check_csrdown
    jsr csrright
    jmp keyloop

check_csrdown
    ; cursor nach unten?
    cmp csrdownkey
    bne check_csrleft
    jsr csrdown
    jmp keyloop

check_csrleft

```

```

; cursor nach links?
cmp csrleftkey
bne check_csrup

jsr csrleft
jmp keyloop

check_csrup
; cursor nach oben?

cmp csrupkey
bne check_quit

jsr csrup
jmp keyloop

check_quit
; editor beenden?

cmp quitkey
beq keyctrl_end

; ansonsten wieder mit der
; tastaturabfrage fortfsetzen

jmp keyloop

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

lda scrcode_clrscr
jsr $ffd2

rts

```

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

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

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

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

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

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

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

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

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

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

```
srcode_bit0
    .byte $2e
srcode_bit1
    .byte $51
```

Ein gelöschtes Bit wird durch einen kleinen Punkt (srcode_bit0) dargestellt, ein gesetztes Bit durch einen großen Punkt (srcode_bit1).

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

```
;-----;
; drawbit
; setzt oder loescht das bit an der
; aktuellen cursorposition
;
; parameter:
; carryflag = 0: bit loeschen
; carryflag = 1: bit setzen
;
; ruckgabewerte:
; keine
;
; aendert:
; a,x,y,status
;
drawbit
    bcc drawbit_bit0
    lda scrcode_bit1
    jmp drawbit_show
drawbit_bit0
    lda scrcode_bit0
drawbit_show
    ldy #$00
    sta ($fb),y
    lda csrcol
    cmp csrmaxcol
    beq drawbit_end
    inc csrcol
    ldx #$01
    ldy #$00
    jsr adc16
drawbit_end
    jsr showcsr
    rts
```

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

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

Sehen wir uns nun das Unterprogramm drawbit an.

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

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

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

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

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

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

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

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

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

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

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

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

```

check_setbit1
; bit setzen?

    cmp setbit1key
    bne check_setbit0

    sec
    jsr drawbit

    jmp keyloop

check_setbit0
; bit loeschen?

    cmp setbit0key
    bne check_quit

    clc
    jsr drawbit

    jmp keyloop

```

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

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

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

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

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

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

Teil 2

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

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

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

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

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

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

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



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

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

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

Sehen Sie sich beispielsweise Reihe 06 an:



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

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

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

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

Machen wir uns daher gleich an die Berechnung dieser Adresse.

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

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

Doch eins nach dem anderen.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Nachfolgend sehen Sie den Assemblercode des Unterprogramms:

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

```

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

idx csrmincol

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

jsr calcposaddr

; bytenummer wieder vom stack
; holen

pla

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

asl a
asl a
asl a

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

sta $fb

lda #\$00
sta $fc

jsr adci6

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

lda $fb
sta $fd

lda $fc
sta $fe

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

pla
sta $fc

pla
sta $fb

rts

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```

;-----  

; hauptprogramm  

; hier wird das programm gestartet  

; start von basic aus mit sys 12288  

* = $3000  

;  

; sichern der parameter,  

; welche vom basic programm im  

; x register und y register  

; abgelegt wurden, weil sie  

; bereits durch den aufruf des  

; unterprogramms init  

; veraendert werden  

txa  

pha  

tya  

pha  

;  

; editor variablen  

; initialisieren  

jsr init  

;  

; benutzeroberflaeche anzeigen  

; jsr draweditor  

;  

; tastaturabfrage starten  

; jsr keyctrl  

;  

; unterprogramm calcbyteaddr  

; testen  

;  

; zuvor gesicherte parameter  

; wieder vom stack holen und  

; inhalte der register x und y  

; wiederherstellen  

pla  

tay  

pla  

tax  

;  

; adresse berechnen  

jsr calcbyteaddr  

rts

```

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

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

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

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

READY.
```

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

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

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

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

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

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

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

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

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

Daraus ergeben sich folgende Parameter für das Unterprogramm calcposaddr:

Zeilenwert: csrminrow + nummer der reihe = 1 + 20 = 21

Spaltenwert: csrmincol = 9

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

Zeilenwert * 40 + Spalte + Startadresse des Bildschirmspeichers

Dies bedeutet in diesem Beispiel hier:

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

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

$$1873 + 16 = 1889$$

Passt also!

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
bytevalue
    .byte $00
```

Nachfolgend sehen Sie den Assemblercode des Unterprogramms getbytefromscr:

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

```

lda powersoftwo,y
clc
adc bytevalue
sta bytevalue

calc_continue
; index auf naechstes zeichen
; setzen

iny
; haben wir bereits alle acht
; zeichen durchlaufen?

cpy #$08
; wenn nicht, dann naechstes
; zeichen pruefen

bne calcloop
; ansonsten sind wir fertig

rts

```

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

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

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

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

Und zweitens beim Laden der zur Bitposition passenden Zweierpotenz:

```
lda powersoftwo,y
```

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

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

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

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

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

Falls nicht, wird das nächste Zeichen geprüft.

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

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

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

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

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

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



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

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

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

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

Hier zum leichteren Vergleich nochmals das Punktmuster:



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

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

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

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

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

```

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

```

```
; print peek (781) geprueft  
; werden  
rts
```

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

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

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

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

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

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

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

Notieren Sie sich das von Ihnen erstellte Punktmuster.

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

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

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

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



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



04.
05.
06.
07.
08.
09.
10.
11.
12.
13.
14.
15.
16.
17.
18.
19.
20.

PRESS SHIFT + H FOR HELP
READY.
PRINT PEEK(781)
217
READY.

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

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

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

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

```
spritedata
    .byte $00,$00,$00
    .byte $00,$00,$00
```

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

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

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

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

Hier das Unterprogramm sprfromscreen:

```
;-----;
; sprfromscr
; wandelt die punktmuster im editor
; in bytewerte um und legt diese im
; feld spritedata ab
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
;
sprfromscr
; wir beginnen bei reihe nr. 0
    lda #$00
    sta rownr
;
; und schreiben die bytes
; ab index 0 in das feld
; spritedata
    sta spritedataindex
;
rowloop
; in jeder reihe beginnen wir
; mit byte nr. 0
    lda #$00
    sta bytenr
;
byteloop
; adresse fuer aktuelles
; byte-punktmuster berechnen
    ldy rownr
    idx bytenr
    jsr calcbyteaddr
;
; punktmuster in bytewert
; umrechnen
    jsr getbytefromscr
;
; errechneten wert ins
; feld spritedata schreiben
```

```

lda bytevalue
ldx spritedataindex
sta spritedata,x

; index fuer spritedata
; erhöhen

inc spritedataindex
; bytenummer erhöhen

inc bytenr

; schon alle bytes in dieser
; reihe bearbeitet?

lda bytenr
cmp #$03

; wenn nicht dann mit
; næchstem byte weitermachen

bne byteloop

; ansonsten weiter zur
; næchsten reihe

; reihennummer erhöhen

inc rownr

; haben wir schon alle reihen
; ausgelesen?

lda rownr
cmp #$15

; wenn nicht, dann weiter zur
; næchsten reihe

bne rowloop

; ansonsten sind wir fertig

rts

```

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

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

Die innere Schleife wird solange durchlaufen, bis der Inhalt der Variablen bytenr nach dem Erhöhen den Wert 3 erreicht hat, wohingegen die äußere Schleife solange durchlaufen wird, bis alle Reihen abgearbeitet wurden.

Dies ist dann der Fall, wenn der Inhalt der Variablen rownr nach dem Erhöhen den Wert 21 annimmt.

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

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

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

Im Unterprogramm keyctrl war folgende Ergänzung nötig:

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

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

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

```
-----  
; savesprite
; uebertraegt das punktmuster aus dem
; editor in das datenfeld spritedata
; und speichert diese 63 bytes in einer
; datei
;  
parameter:
; keine
;  
rueckgabewerte:
; keine
;  
aendert:
; a,x,y,status
;  
savesprite
    ; cursor ausblenden
    jsr removecsr
    ; punktmuster ins feld
    ; spritedata uebertragen
    jsr sprfromscr
    ; cursor wieder einblenden
    jsr showcsr
    ; adresse des feldes
    ; spritedata im x register
    ; und y register ablegen
    ; damit das feld von basic
    ; aus ausgelesen werden kann
    ldx #<spritedata
    ldy #>spritedata
    rts
```

Hier wird zunächst der Cursor aus dem erwähnten Grund ausgeblendet.

Dann wird das Punktmuster aus dem Editor in das Datenfeld spritedata übertragen und der Cursor wieder eingeblendet.

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

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

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

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

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

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

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

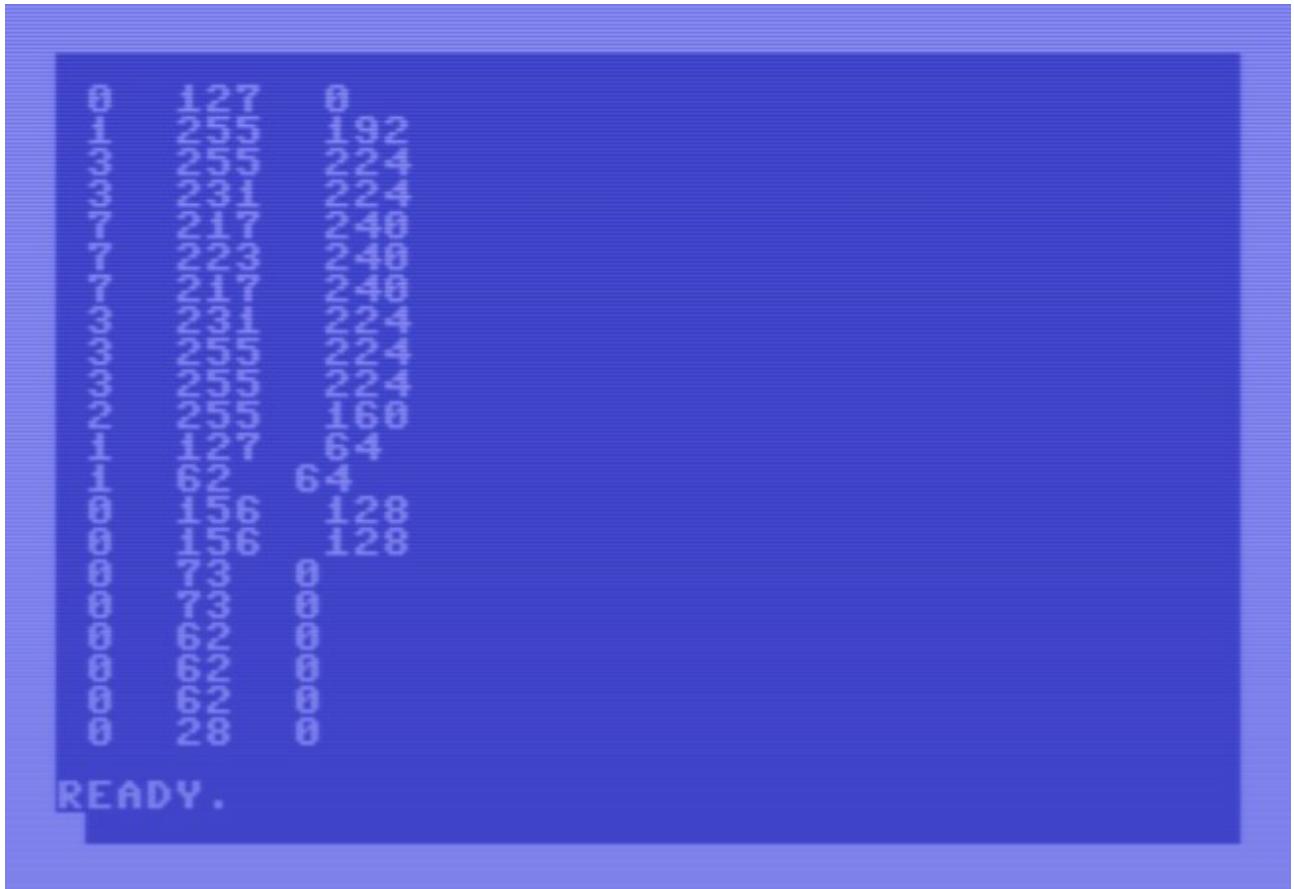
Das Programm löscht zunächst den Bildschirm und liest den Inhalt des X Registers und des Y Registers aus den Speicherstellen 781 und 782. Das Unterprogramm savesprite hat dort vor seiner Beendigung das niederwertige und höherwertige Byte der Adresse des Feldes spritedata abgelegt.

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

Nun hat das Basic-Programm direkten Zugriff auf das Feld spritedata.
Es liest die 63 Bytes aus dem Feld und gibt diese in 21 Reihen zu je 3 Bytes am Bildschirm aus.

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

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



| Byte Nr. 0 | Byte Nr. 1 | Byte Nr. 2 |
|------------|------------|------------|
| 0 | 127 | 0 |
| 1 | 255 | 192 |
| 3 | 255 | 224 |
| 3 | 231 | 224 |
| 7 | 217 | 240 |
| 7 | 223 | 240 |
| 7 | 217 | 240 |
| 3 | 231 | 224 |
| 3 | 255 | 224 |
| 3 | 255 | 224 |
| 2 | 255 | 160 |
| 1 | 127 | 64 |
| 1 | 62 | 64 |
| 0 | 156 | 128 |
| 0 | 156 | 128 |
| 0 | 73 | 0 |
| 0 | 73 | 0 |
| 0 | 62 | 0 |
| 0 | 62 | 0 |
| 0 | 62 | 0 |
| 0 | 28 | 0 |

| Byte Nr. 0 | Byte Nr. 1 | Byte Nr. 2 |
|------------|------------|------------|
| 0 | 156 | 128 |
| 0 | 156 | 128 |
| 0 | 73 | 0 |
| 0 | 73 | 0 |
| 0 | 62 | 0 |
| 0 | 62 | 0 |
| 0 | 62 | 0 |
| 0 | 28 | 0 |

Dateioperationen in Assembler

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

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

Es besteht aus folgenden Unterprogrammen:

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

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

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

```

filename .text "sprite,seq"
scratchfilename .text "s:sprite"

```

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

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

Das Programm führt nun folgende Schritte aus:

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

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

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

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

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

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

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

In Basic würde das so aussehen:

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

Und hier die Umsetzung in das Assembler-Unterprogramm scratchfile:

```
;-----  
; scratchfile  
; loescht die datei namens sprite  
;  
; parameter:  
; keine  
;  
; rueckgabewerte:  
; keine  
;  
; aendert:  
; a,x,y,status  
scratchfile  
; setfls  
lda #$01  
idx #$08  
ldy #$0f  
jsr $ffba  
; setnam  
lda #$08  
idx #<scratchfilename  
ldy #>scratchfilename  
jsr $ffbd  
; open  
jsr $ffc0  
; close  
lda #$01  
jsr $ffc3  
rts
```

Zuallererst wird immer die Kernal-Funktion SETLFS aufgerufen.

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

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

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

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

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

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

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

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

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

Sehen Sie sich dazu das Assembler-Unterprogramm sprdatatofile an:

```
;-----  
; sprdatatofile  
; schreibt die bytes aus dem datenfeld  
; spritedata in eine sequentielle datei  
; namens sprite  
  
; parameter:  
; keine  
  
; rueckgabewerte:  
; keine  
  
; aendert:  
; a,x,y,status  
  
sprdatatofile  
    ; setfis  
        lda #$01  
        idx #$08  
        idy #$01  
        jsr $ffba  
        ; setnam  
        lda #$0a  
        idx #<filename  
        idy #>filename  
        jsr $ffbd  
        ; open  
        jsr $ffc0  
        ; chkout  
        idx #$01  
        jsr $ffc9  
        ; bytes in datei schreiben  
        idx #$00  
  
writeloop  
    ; byte aus spritedata holen  
    lda spritedata,x  
    ; byte in datei schreiben  
    jsr $ffd2  
    ; index auf naechstes byte  
    inx  
    ; bereits alle bytes  
    ; geschrieben?  
    cpx #$3f
```

```

; falls nicht, dann weiter
; zum naechsten byte

bne writeloop

; clrch
jsr $ffcc

; close
lda #$01
jsr $ffc3

rts

```

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

OPEN 1,8,1,“SPRITE,SEQ“

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

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

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

Wir haben die Funktion CHROUT (\$FFD2) schon öfters für die Ausgabe von Zeichen auf dem Bildschirm aufgerufen. Sie wird ebenfalls genutzt, um Zeichen in eine Datei zu schreiben.

In diesem Fall ist jedoch ein vorheriger Aufruf der Funktion CHKOUT nötig, um über die Angabe der logischen Dateinummer die Datei auszuwählen, in die diese Zeichen geschrieben werden sollen.

Ab diesem Zeitpunkt landen alle über den Aufruf der Funktion CHROUT ausgegebenen Zeichen nicht mehr am Bildschirm, sondern in dieser Datei.

Nun werden in einer Schleife namens writeloop nacheinander alle Bytes des Datenfeldes spritedata in den Akkumulator geladen. Da sich die Funktion CHROUT den Zeichencode des zu schreibenden Zeichens aus dem Akkumulator holt, kann sie darauffolgend direkt ohne Umschweife aufgerufen werden, wodurch das jeweilige Zeichen in der Datei landet.

Danach wird der Index durch den Befehl INX auf das nächste Byte gesetzt und geprüft, ob schon alle Bytes aus dem Datenfeld spritedata verarbeitet wurden. Falls dies nicht der Fall sein sollte, wird wieder zum Label writeloop gesprungen und das nächste Byte verarbeitet.

Ansonsten wird durch den Aufruf der Kernal-Funktion CLRCH der Ausgabekanal wieder auf den Bildschirm eingestellt, wodurch die Ausgaben ab nun nicht mehr in die Datei, sondern wieder auf dem Bildschirm ausgegeben werden.

Zum Schluss wird die Datei noch geschlossen, was durch den Aufruf der Funktion CLOSE und vorherigem Laden der logischen Dateinummer in den Akkumulator erfolgt.

So, nun haben wir unsere Spritedaten in die Datei geschrieben und nun wollen wir sie von dort wieder in das Datenfeld spritedata holen.

Hier sehen Sie den Assemblercode des Unterprogramms sprdatafromfile:

```
;-----  
; sprdatafromfile  
; laedt die spritedaten aus einer  
; sequentiellen datei namens sprite  
; in das datenfeld spritedata  
  
; parameter:  
; keine  
  
; rueckgabewerte:  
; keine  
  
; aendert:  
; a,x,y,status  
  
sprdatafromfile  
    ; setfls  
        lda #$01  
        ldx #$08  
        ldy #$00  
        jsr $ffba  
  
    ; setnam  
        lda #$0a  
        ldx #<filename  
        ldy #>filename  
        jsr $ffbd  
  
    ; open  
        jsr $ffc0  
  
    ; chkin  
        ldx #$01  
        jsr $ffc6  
  
    ; bytes aus datei lesen  
        ldx #$00  
  
readloop  
    ; byte aus datei lesen  
        jsr $ffcf  
  
    ; gelesenes byte in feld  
    ; spritedata schreiben  
        sta spritedata,x  
  
    ; index auf naechstes byte  
        inx  
  
    ; bereits alle bytes gelesen?  
        cpx #$3f  
  
    ; falls nicht, dann weiter  
    ; zum naechsten byte  
        bne readloop  
  
    ; clrch  
        jsr $ffcc  
  
    ; close  
        lda #$01  
        jsr $ffc3
```

Die Aufrufe von SETFLS, SETNAM und OPEN entsprechen folgendem Basic-Befehl:

OPEN 1,8,0,“SPRITE,SEQ“

Analog zu vorhin, müssen wir als nächstes unsere Datei als Eingabekanal festlegen, da normalerweise Eingaben ja von der Tastatur gelesen werden und nicht aus einer Datei.

Dies geschieht durch Aufruf der Kernal-Funktion CHKIN, welche die logische Dateinummer im X Register erwartet.

Von nun an wird nicht mehr von der Tastatur, sondern aus unserer Datei gelesen.

Das gelesene Byte wird an die durch den aktuellen Index bestimmte Stelle im Datenfeld spritedata geschrieben.

Dann wird der Index durch den Befehl INX auf das nächste Byte gesetzt und geprüft, ob bereits alle 63 Bytes aus der Datei gelesen wurden. Falls dies nicht der Fall sein sollte, wird wieder zum Label readloop gesprungen und das nächste Byte verarbeitet.

Ansonsten wird durch den Aufruf der Kernal-Funktion CLRCH der Eingabekanal wieder auf die Tastatur eingestellt, wodurch die Eingaben ab nun nicht mehr aus der Datei, sondern wieder von der Tastatur entgegengenommen werden.

Hinweis:

Die Funktion CLRCH setzt sowohl den Standard-Ausgabekanal als auch den Standard-Eingabekanal zurück auf den Bildschirm bzw. die Tastatur.

Zum Schluss wird wie vorhin die Datei noch geschlossen, was durch den Aufruf der Funktion CLOSE und vorherigem Laden der logischen Dateinummer in den Akkumulator erfolgt.

Das Unterprogramm showsprite ist nur hier in diesem Programm von Bedeutung. Es erzeugt aus den Spritedaten im Datenfeld spritedata ein Sprite und stellt es am Bildschirm dar. In diesem Fall wird in der linken oberen Ecke ein gelber Ballon angezeigt. Auf diese Weise können wir kontrollieren, ob alle Bytes korrekt geschrieben und gelesen wurden.

```
;-----
; showsprite
; zeigt das sprite dessen aussehen
; durch das feld spritedata definiert
; wird am bildschirm an
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
;
showsprite
; block ii fuer das sprite
; auswaehlen
```

```

    lda #$0b
    sta $07f8

    ; bytes aus dem datenfeld
    ; spritedata in diesen block
    ; kopieren

    ldx #$00

copyloop ; byte kopieren
    lda spritedata,x
    sta $02c0,x

    ; index auf naechstes byte
    inx

    ; bereits alle bytes kopiert?
    cpx #$3f

    ; falls nicht, dann weiter
    ; zum naechsten byte
    bne copyloop

    ; sprite ist einfarbig
    lda #$00
    sta $d01c

    ; sprite soll gelb sein
    lda #$07
    sta $d027

    ; spriteposition festlegen
    ; x = $18 (24), y = $32 (50)
    lda #$18
    sta $d000

    lda #$32
    sta $d001

    ; x ist <= 255, daher setzen
    ; wir das erweiterungsbit
    ; fuer die x koordinate
    ; zurueck
    lda #$00
    sta $d010

    ; sprite hat prioritaet vor
    ; dem hintergrund
    lda #$00
    sta $d01b

    ; sprite aktivieren
    lda #$01
    sta $d015

    rts

```

Hier begegnet uns nichts neues, denn all dies kennen wir bereits aus dem Kapitel zum Thema Sprites.

Was jedoch zu erwähnen wäre:

Da wir es hier in diesem Programm ja sowieso nur mit einem einzigen Sprite, dem Sprite 0, zu tun haben, habe ich auf das Maskieren der Bits beim Setzen der Werte verzichtet.

Sehen Sie z.B. den letzten Abschnitt oben an:

```
; sprite aktivieren
lda #$01
sta $d015
rts
```

Dieser Befehl aktiviert Sprite 0 und deaktiviert gleichzeitig die Sprites 1 bis 7. Da wir aber nur mit Sprite 0 arbeiten, ist es für uns nicht von Bedeutung, dass die Sprites 1 bis 7 deaktiviert werden.

Ein anderes Beispiel ist dieser Abschnitt hier:

```
; sprite ist einfarbig
lda #$00
sta $d01c
```

Auch hier wird keine Rücksicht auf die Sprites 1 bis 7 genommen, sondern es werden alle 8 Sprites als einfarbig definiert. Dies ist jedoch für uns nicht von Interesse, da wir ja nur mit dem Sprite 0 arbeiten.

Werfen wir abschließend noch einen Blick auf den Hauptteil des Programms.

```
-----
; hauptprogramm
; hier wird das programm gestartet
; start von basic aus mit sys 12288
*= $3000
; sprite ausblenden
lda #$00
sta $d015
; datei loeschen
jsr scratchfile
; spritedaten in datei
; schreiben
jsr spridataToFile
; spritedaten loeschen
jsr clrspriteData
; spritedaten aus datei
; laden
jsr spridataFromFile
; sprite anzeigen
jsr showsprite
; adresse von spritedata
```

```
; im x register und y register
; hinterlegen, damit das feld
; von basic aus ausgelesen
; werden kann

ldx #<spritedata
ldy #>spritedata

rts
```

Zu Beginn wird das Sprite 0 (und auch alle anderen, was jedoch hier wie gesagt nicht von Bedeutung ist) ausgeblendet, denn es kann ja sein, dass das Sprite durch einen vorausgegangenen Aufruf des Programms noch am Bildschirm angezeigt wird.

Dann wird die Datei namens SPRITE gelöscht. Dann werden die Bytes aus dem Datenfeld spritedata in die neu angelegte Datei gleichen Namens gespeichert.

Nun werden die 63 Bytes aus der Datei namens SPRITE in das Datenfeld spritedata eingelesen und durch der Aufruf des Unterprogramms showsprite erzeugt das Sprite basierend auf diesen Daten und zeigt es am Bildschirm an.

Abschließend wird die noch das niederwertige Byte der Adresse des Datenfeldes spritedata in das X Register und das höherwertige Byte der Adresse in das Y Register geschrieben.

Dadurch hat man die Möglichkeit, von Basic aus auf die Spritedaten zuzugreifen und kann sich diese eventuell nochmal zur Kontrolle ausgeben lassen.

Nachdem Sie das Programm SPRITEIO durch Eingabe von SYS 12288 gestartet haben, sollten Sie nach einigen Sekunden folgendes am Bildschirm sehen:



Blenden Sie nun das Sprite durch Eingabe des Befehls POKE 53269,0 aus und laden das Basic-Programm SHOWSPRBYTES.

```
READY.  
SYS 12288  
READY.  
POKE 53269,0  
READY.  
LOAD "SHOWSPRBYTES",8  
SEARCHING FOR SHOWSPRBYTES  
LOADING  
READY.  
LIST  
10 A=PEEK(781)+256*PEEK(782)  
20 FOR R=0 TO 20  
30 FOR B=0 TO 2  
40 PRINT PEEK(A+R*3+B);  
50 NEXT B  
60 PRINT  
70 NEXT R  
READY.
```

Das Programm bildet aus dem nieder- und höherwertigen Byte der Adresse des Datenfeldes spritedata wieder die volle Adresse und speichert diese in der Variablen A.

Dann werden die 63 Bytes aus dem Datenfeld spritedata in zwei ineinander verschachtelten Schleifen ausgelesen und in Gruppen zu je 3 Bytes untereinander ausgegeben.

Nach Eingabe von RUN sollten nun die 21 Reihen mit jeweils 3 Bytes angezeigt werden.

```

RUN
8 127 8
1 255 192
3 255 224
3 255 224
3 231 224
7 217 240
7 223 240
7 217 240
3 231 224
3 255 224
3 255 224
2 255 160
1 127 64
1 62 64
0 156 128
0 156 128
0 73 0
0 73 0
0 62 0
0 62 0
0 62 0
0 28 0

READY .

```

Ok, soviel zu den elementaren Dateioperationen in Assembler.

Sehen wir uns nun an, wie ich die vorgestellten Unterprogramme zum Speichern und Laden der Spritedaten in den Sprite-Editor integriert habe.

Die Unterprogramme scratchfile, sprdatatypefile und sprdatafromfile existieren 1:1 identisch auch im Sprite-Editor.

Damit das Speichern und Laden der Spritedaten über die entsprechenden Tastenkombinationen auch registriert wird, muss das Unterprogramm keyctrl um folgenden Abschnitt ergänzt werden:

```

check_save
; sprite speichern?
    cmp savekey
    bne check_load
    jsr savesprite
    jmp keyloop

check_load
; sprite laden?
    cmp loadkey
    bne check_quit
    jsr loadsprite
    jmp keyloop

```

Und die Abfrage unmittelbar davor muss ebenfalls an das neue Label check_save angepasst werden:

```
check_setbit0
; bit loeschen?

cmp setbit0key
bne check_save

clc
jsr drawbit
jmp keyloop
```

Neu sind auch die beiden Unterprogramme savesprite und loadsprite:

```
;-----
; savesprite
; uebertraegt das punktmuster aus dem
; editor in das datenfeld spritedata
; und speichert diese 63 bytes in einer
; datei

; parameter:
; keine

; rueckgabewerte:
; keine

; aendert:
; a,x,y,status

savesprite
; rahmenfarbe fuer die dauer
; des speichervorgangs aendern
lda #$0d
sta $d020

; cursor ausblenden
jsr removecsr

; punktmuster ins feld
; spritedata uebertragen
jsr sprfromscr

; cursor wieder einblenden
jsr showcsr

; datei loeschen
jsr scratchfile

; spritedaten in datei
; schreiben
jsr sprdatatofile

; rahmenfarbe wieder auf
; vorherige farbe einstellen
lda #$0e
sta $d020

rts
```

Hier erkennt man die einzelnen Schritte wieder, die beim Speichern der Spritedaten durchlaufen wurden. Neu hinzugekommen ist jedoch die Änderung der Rahmenfarbe für die Dauer des Speichervorgangs.

Um dem Benutzer optisch anzuzeigen, dass die Spritedaten gerade in die Datei geschrieben werden, ändert sich die Rahmenfarbe nach dem Drücken der Tastenkombination SHIFT + S.

Der Rahmen wird solange in dieser Farbe angezeigt, bis der Speichervorgang abgeschlossen ist und wenn dies der Fall ist, erscheint der Rahmen wieder in der vorherigen Farbe.

Das Unterprogramm loadsprite ist noch nicht fertig, es werden hier zum jetzigen Zeitpunkt nur die Spritedaten aus der Datei in das Datenfeld spritedata geladen.

```
;-----  
; loadsprite  
; laedt die spritedaten aus der datei  
; namens sprite ins datenfeld  
; spritedata und gibt diese wieder  
; als punktmuster am bildschirm aus  
  
; parameter:  
; keine  
  
; rueckgabewerte:  
; keine  
  
; aendert:  
; a,x,y,status  
  
loadsprite  
    ; spritedaten aus datei laden  
    jsr sprdatafromfile  
    rts
```

Angenommen, wir haben mit dem aktuellen Stand des Sprite-Editors ein Sprite erstellt, dieses in der Datei SPRITE gespeichert und wollen dessen Aussehen irgendwann nun ändern.

Dazu laden wir mit dem Unterprogramm sprdatafromfile die Spritedaten aus der Datei in das Datenfeld spritedata.

Das Problem: Dort können wir sie nicht sehen, d.h. wir müssen nun den umgekehrten Weg beschreiten und die Bytes aus dem Datenfeld wieder in Punktmuster umwandeln und im Sprite-Editor anzeigen.

Nehmen wir an, dass sich an irgendeiner Stelle im Datenfeld spritedata der Wert 217 befindet.

Dies entspricht dem binären Wert %11011001, was am Bildschirm dem Punktmuster



Wir brauchen also nun ein Unterprogramm, das uns ausgehend von einem Byte aus dem Datenfeld spritedata wieder das entsprechende Punktmuster auf dem Bildschirm bringt.

Doch bevor wir ein Punktmuster am Bildschirm ausgeben können, müssen wir zuerst einmal wissen, wo wir es ausgeben müssen, besser gesagt an welcher Adresse im Bildschirmspeicher.

In den 8 Speicherstellen ab dieser Startadresse wird dann das Punktmuster Zeichen für Zeichen im Bildschirmspeicher wieder aufgebaut.

Falls in unserem Byte, dessen Punktmuster wir wieder auf den Bildschirm bringen wollen, ein Bit gesetzt ist, schreiben wir an den entsprechenden Positionen den Screencode für den großen Punkt in den Bildschirmspeicher und andernfalls den Screencode für den kleinen Punkt.

Aber wie prüfen wir, ob in dem Byte ein Bit an einer bestimmten Position gesetzt ist?

Dies lässt sich mit einer UND-Verknüpfung zwischen dem Byte und der Zweierpotenz, welche dieser Bitposition entspricht, herausfinden.

Nehmen wir als Beispiel wieder das genannte Byte mit dem Inhalt %11011001.

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|-----|----|----|----|---|---|---|---|
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 1 |

Die erste Zeile enthält die Nummern der Bitpositionen, die zweite Zeile die jeweilige Zweierpotenz an dieser Position und die dritte Zeile die Bits unserer Zahl, wobei die Stellen an denen eine 1 steht, grün markiert sind.

Um die UND-Verknüpfungen nach der Reihe für jede Bitposition durchführen zu können, legen wir uns im Datenabschnitt ein Feld mit den benötigten Zweierpotenzen an. Wichtig ist hierbei die Reihenfolge, wir beginnen bei 128 (\$80) und enden bei 1 (\$01).

```
powersoftwo
:byte $80,$40,$20,$10
:byte $08,$04,$02,$01
```

Spielen wir den Aufbau des Punktmusters im Bildschirmspeicher mal anhand eines Beispiels losgelöst vom Assembler-Code durch.

Angenommen, wir zählen im Y Register einen Index von 0 bis 7 hoch. Dieser Index soll sowohl als Index in das Datenfeld powersoftwo als auch als Index für die indirekte Y nachindizierte Adressierung bezogen auf die Startadresse des Punktmusters im Bildschirmspeicher dienen.

Vorhin habe ich erwähnt, das wir zuerst diese Startadresse ausrechnen müssen, bevor wir mit der Ausgabe des Punktmusters beginnen können.

Nehmen wir an, diese Berechnung hätte die Adresse 1100 (\$044C) ergeben.

| Y Register | Zweierpotenz | UND-Verknüpfung | Ergebnis der UND-Verknüpfung | Adresse im Bildschirmspeicher (1100),Y | Auszugebendes Zeichen |
|------------|--------------------|------------------------|------------------------------|--|-----------------------|
| 0 | powersoftwo,0 =128 | %11011001 %10000000 | %10000000 | 1100 + 0 = 1100 | █ |
| 1 | powersoftwo,1 =64 | %11011001 %01000000 | %01000000 | 1100 + 1 = 1101 | █ |
| 2 | powersoftwo,2 =32 | %11011001 %00100000 | %00000000 | 1100 + 2 = 1102 | █ |
| 3 | powersoftwo,3 =16 | %11011001 %00010000 | %00010000 | 1100 + 3 = 1103 | █ |
| 4 | powersoftwo,4 =8 | %11011001 | %00001000 | 1100 + 4 = 1104 | █ |

| Y Register | Zweierpotenz | UND-Verknüpfung | Ergebnis der UND-Verknüpfung | Adresse im Bildschirmspeicher (1100),Y | Auszgebendes Zeichen |
|------------|------------------|-------------------------|------------------------------|--|----------------------|
| %000001000 | | | | | |
| 5 | powersoftwo,5 =4 | %11011001 %000000100 | %000000000 | 1100 + 5 = 1105 | █ |
| 6 | powersoftwo,6 =2 | %11011001 %000000010 | %000000000 | 1100 + 6 = 1106 | █ |
| 7 | powersoftwo,7 =1 | %11011001 %000000001 | %000000001 | 1100 + 7 = 1107 | █ |

Und hier die Umsetzung des in der Tabelle dargestellten Ablaufs in Assembler-Code:

```
;-----  
; putbytetoscr  
; wandelt ein byte in ein achstelliges  
; Punktmuster um und gibt dieses in  
; der uebergebenen reihe und "byte" des  
; sprites aus  
  
; parameter:  
; bytewert: in variablen bytevalue  
; reihennr (0 bis 20): y register  
; bytenr (0,1 oder 2): x register  
  
; ruckgabewerte:  
; keine  
  
; aendert:  
; a,x,y,status  
  
putbytetoscr  
    ; adresse des punktmusters im  
    ; bildschirmspeicher berechnen  
        jsr calcbyteaddr  
        ldy #$00  
  
printloop  
    lda powersoftwo,y  
    ; bit gesetzt?  
    and bytevalue  
    beq bit_notset  
  
bit_set  
    lda srccode_bit1  
    jmp print_continue  
  
bit_notset  
    lda srccode_bit0  
  
print_continue  
    sta ($fd),y  
    iny  
    cpy #$08  
    bne printloop  
  
rts
```

Zuerst wird anhand der übergebenen Reihen- und Bytenummer die Adresse berechnet, an der das entsprechende Punktmuster im Bildschirmspeicher beginnt.

Nun beginnt der Ablauf, welcher in der obigen Tabelle dargestellt ist.

Die zur Bitposition passende Zweierpotenz wird aus dem Datenfeld powersoftwo in den Akkumulator geladen.

Das Byte, welches als Punktmuster dargestellt werden soll, wurde in der Variablen bytevalue als Parameter an das Unterprogramm übergeben.

Nun wird durch die UND-Verknüpfung festgestellt, ob das jeweilige Bit gesetzt ist oder nicht.

Wenn nicht, wird zum Label bit_notset gesprungen, an dem der Screencode für den kleinen Punkt in den Akkumulator geladen wird.

Wenn ja, wird der Screencode für den großen Punkt in den Akkumulator geladen.

Nun wird der Screencode durch den Befehl STA (\$FD),Y an die jeweilige Position im Bildschirmspeicher geschrieben.

Dies wiederholt sich für alle acht Bitpositionen bis schlussendlich das vollständige, achtstellige Punktmuster am Bildschirm zu sehen ist.

Mit dem Unterprogramm putbytetoscr kann man also Bytes in Punktmuster umwandeln und basierend auf der Reihen- und Bytenummer auch an der korrekten Position am Bildschirm darstellen.

Was nun noch zur Ausgabe fehlt, ist ein Unterprogramm, welches Byte für Byte das komplette Datenfeld spritedata durchläuft, in Punktmuster umwandelt und diese eines nach dem anderen am Bildschirm ausgibt.

Wir nennen es sprtoscr und hier sehen Sie dessen Assembler-Code:

```
;-----  
; sprtoscr  
; wandelt die bytes im datenfeld  
; spritedata in die entsprechenden  
; Punktmuster auf dem Bildschirm um  
;  
; Parameter:  
; keine  
;  
; Rueckgabewerte:  
; keine  
;  
; aendert:  
; a,x,y,status  
  
sprtoscr    lda #$00  
            sta rownr  
            sta spritedataindex  
  
row_loop    lda #$00  
            sta bytenr  
  
byte_loop   idx spritedataindex  
            lda spritedata,x  
            sta bytevalue  
  
            idx bytenr  
            ldy rownr
```

```

jsr putbytetoscr
inc spritedataindex
inc bytenr
lda bytenr
cmp #$03
bne byte_loop

inc rownr
lda rownr
cmp #$15
bne row_loop

rts

```

Das Unterprogramm durchläuft das Datenfeld spritedata Byte für Byte und ruft für jedes Byte das zuvor erstellte Unterprogramm putbytetoscr auf.

Die Variable rownr enthält dabei die jeweilige Reihennummer und die Variable bytenr die jeweilige Bytenummer für den Aufruf von putbytetoscr.

Die Variable spritedataindex wird während des Ablaufs des Unterprogramms sprtoscr von 0 bis 62 hochgezählt und dient als Index im X Register, um sich mit dem Befehl LDA SPRITEDATA,X ein Byte nach dem anderen aus dem Datenfeld spritedata zu holen.

Es sind hier zwei ineinander verschachtelte Schleifen zu erkennen. Eine äußere Schleife, welche über wiederholtes Springen zum Label row_loop die Reihen 0 bis 20 abarbeitet und eine innere Schleife, welche die drei Bytes 0 bis 2 in jeder Reihe bearbeitet und über wiederholtes Springen zum Label byte_loop durchlaufen wird.

Als letzten Schritt müssen wir noch den Cursor an die Ausgangsposition, also in die linke obere Ecke des Editorbereichs versetzen.

Dazu habe ich folgendes kleines Unterprogramm namens csrhome geschrieben:

```

;-----
; csrhome
; setzt den cursor an die
; ausgangsposition, also in die linke
; obere ecke des editorbereichs
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,y,status
;
csrhome
    ldx csrhomeaddr
    ldy csrhomeaddr+1
    stx $fb
    sty $fc

    lda csrminalrow
    sta csrrrow

    lda csrminalcol
    sta csrcol

    jsr showcsr

```

Beim Starten des Editors wurde in der Variablen csrhomeaddr jene Adresse im Bildschirmspeicher hinterlegt, die der Ausgangsposition des Cursors, also der linken oberen Ecke des Editorbereichs entspricht.

Das Unterprogramm csrhome tut nun nichts anderes, als genau diese Adresse in die Speicherstellen \$FB und \$FC zu transportieren und die Variablen csrrow und csrcol ebenfalls wieder mit jenen Werten zu versorgen, welche den Koordinaten der Ausgangsposition des Cursors entsprechen.

Nun können wir das Unterprogramm loadsprite vervollständigen:

```
;-----  
; loadsprite  
; laedt die spritedaten aus der datei  
; namens sprite ins datenfeld  
; spritedata und gibt diese wieder  
; als punktmuster am bildschirm aus  
  
; parameter:  
; keine  
  
; rueckgabewerte:  
; keine  
  
; aendert:  
; a,x,y,status  
  
loadsprite  
    ; rahmenfarbe fuer die dauer  
    ; des ladevorgangs aendern  
  
    lda #$0d  
    sta $d020  
  
    ; spritedaten aus datei laden  
    jsr sprdatafromfile  
  
    ; spritedaten in punktmuster  
    ; am bildschirm umwandeln  
    jsr sprtoscr  
  
    ; cursor an die  
    ; ausgangsposition setzen  
    jsr csrhome  
  
    ; rahmenfarbe wieder auf  
    ; vorherige farbe einstellen  
    lda #$0e  
    sta $d020  
  
    rts
```

Analog zum Speichern der Spritedaten ändert sich auch während des Ladens der Spritedaten die Rahmenfarbe, um damit anzuzeigen, dass der Ladevorgang aktuell im Gange ist.

Nachdem die Spritedaten aus der Datei ins Datenfeld spritedata transportiert wurden, werden sie durch den Aufruf des Unterprogramms sprtoscr wieder in Punktmuster auf dem Bildschirm verwandelt.

Dann wird der Cursor in die linke obere Ecke des Editorbereichs versetzt und der Rahmen wieder in der ursprünglichen Farbe angezeigt.

Nun fehlt uns nur noch eine einzige Editorfunktion, nämlich die Möglichkeit zum Löschen des Editorbereichs.

Dazu habe ich zunächst das Unterprogramm clrspritedata geschrieben:

```
;-----  
; clrspritedata  
; setzt alle bytes im datenfeld  
; spritedata auf den wert 0, loescht  
; also das spritemuster im speicher  
  
; parameter:  
; keine  
  
; rueckgabewerte:  
; keine  
  
; aendert:  
; a,x,status  
  
clrspritedata  
    lda #$00  
    ldx #$00  
  
cirloop      sta spritedata,x  
    inx  
    cpx #$3f  
    bne cirloop  
  
    rts
```

Wie sie hier sehen können, durchläuft es in einer Schleife alle 63 Bytes des Datenfeldes spritedata und setzt diese auf den Wert 0.

Den gesamten Vorgang des Löschens verfrachten wir in das Unterprogramm clrsprite:

```
;-----  
; clrsprite  
; loescht den editorbereich  
  
; parameter:  
; keine  
  
; rueckgabewerte:  
; keine  
  
; aendert:  
; a,x,status  
  
clrsprite      ; spritedaten loeschen  
    jsr clrspritedata  
    ; leeres sprite im editor  
    ; anzeigen  
    jsr sprtoscr  
    ; cursor an die  
    ; ausgangsposition setzen  
    jsr csrhome  
  
    rts
```

Durch den Aufruf des soeben definierten Unterprogramms clrsprite data werden die Bytes des Datenfelds sprite data auf den Wert 0 gesetzt.

Dann werden alle 63 Bytes (welche nun den Wert 0 enthalten) in Punktmuster auf den Bildschirm verwandelt, wodurch wieder eine leere Editorfläche bestehend aus kleinen Punkten angezeigt wird.

Anschließend wird der Cursor noch an die Ausgangsposition versetzt und damit haben wir auch diese Programmfunction erfolgreich umgesetzt.

Nun müssen wir nur noch den entsprechenden Aufruf in das Unterprogramm keyctrl einbauen.

```
check_load      ; sprite laden?
    cmp loadkey
    bne check_clear
    jsr loadsprite
    jmp keyloop

check_clear     ; sprite loeschen?
    cmp clearkey
    bne check_quit
    jsr clrsprite
    jmp keyloop
```

Im Abschnitt nach dem Label check_load müssen wir nun anstelle des Labels check_quit das Label check_clear anspringen. Hat der Benutzer die Tastenkombination SHIFT + CLR/HOME gedrückt, wird durch den Befehl JSR CLRSPRITE das Unterprogramm clrsprite aufgerufen und der Editorbereich gelöscht.

Kommen wir nun zur allerletzten Programmfunction, der Anzeige eines Fensters mit Hilfsinformationen, welche über die Tastenkombination SHIFT + H aufgerufen wird.

Das Fenster soll folgendermaßen angezeigt werden:

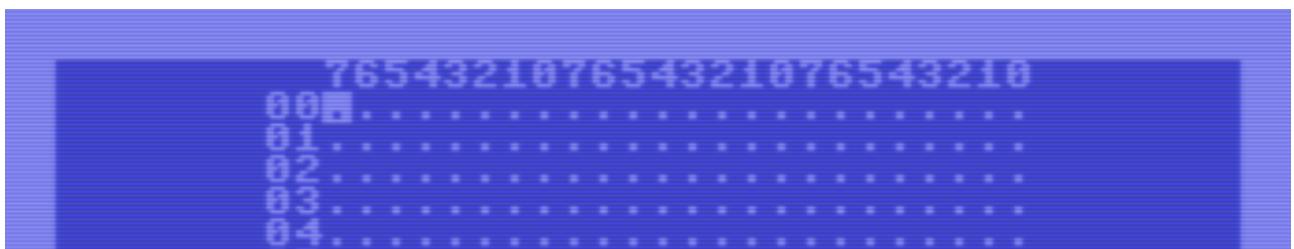


Geschlossen wird das Fenster durch einen beliebigen Tastendruck.

Sehen wir uns den grundsätzlichen Ablauf an.

Bevor das Fenster eingeblendet wird, müssen wir den Inhalt der obersten 6 Zeilen sichern, da nach dem Schließen des Fensters ja wieder der ursprüngliche Inhalt dieser 6 Zeilen sichtbar werden soll.

Ansonsten würde uns folgender Teil des Bildschirmspeichers verloren gehen, da dieser durch die Anzeige des Fensters überschrieben wird.



Um den Inhalt dieser 6 Zeilen zu sichern, müssen wir uns im Datenabschnitt ein Datenfeld definieren, das groß genug ist, um die Zeichen in diesem Bereich aufnehmen zu können.

Wie groß muß dieses Feld sein? Jede Bildschirmzeile besteht aus 40 Zeichen, das ergibt bei 6 Zeilen also 240 Bytes.

Warum ich das Fenster auf 6 Zeilen begrenzt habe, werde ich im Anschluss noch erläutern.

Drückt der Benutzer also die Tastenkombination SHIFT + H, wird zunächst der Inhalt dieser 6 Zeilen in dieses Datenfeld kopiert.

Dann werden die Zeilen gelöscht und das Fenster mit den Hilfsinformation wird eingeblendet. Die Zeilen müssen wir deswegen löschen, da sich der Inhalt des Fensters ansonsten mit dem vorherigen Inhalt der Zeilen vermischen würde.

Das Fenster bleibt solange sichtbar, bis der Benutzer eine beliebige Taste drückt.

Nun muss der ursprüngliche Inhalt der 6 Zeilen wiederhergestellt werden. Dazu kopieren wir die 240 Bytes, die wir zuvor im Datenfeld gesichert haben wieder in den Bildschirmspeicher.

Warum habe ich den Umfang des Fensters auf 6 Zeilen begrenzt? Die Ursache ist in der indizierten Adressierung zu finden.

Die Indizierung findet entweder über das X Register oder das Y Register statt. Da diese Register nur ein Byte fassen können, liegt der maximale Index also bei 255. Wir können mittels der indizierten Adressierung also nur einen Speicherbereich ansprechen, der maximal 256 Bytes umfasst.

Und da das Sichern, Löschen und Wiederherstellen der Zeilen durch indizierte Adressierung stattfindet, sind wir hier auf eine maximale Größe von 256 begrenzt.

Bereits ab 7 Zeilen würden wir den Maximalwert überschreiten, da 7 Zeilen bereits 280 Bytes umfassen.

Natürlich gibt es Möglichkeiten, größere Bereiche des Bildschirms anzusprechen. Man müsste dann allerdings eine Logik umsetzen, die den Bereich in mehrere Teile zu je 256 Bytes, also in Pages, unterteilt und den Zugriff entsprechend über die nummerierten Pages realisieren.

Ich wollte den Sprite-Editor jedoch nicht noch komplizierter machen und deshalb habe ich das Hilfefenster auf 6 Zeilen begrenzt, damit wir in Bezug auf den Index mit einem Byte auskommen.

Beginnen wir mit der Definition des Datenfeldes, welches zur Sicherung der 6 Bildschirmzeilen dient. Ich habe das Datenfeld screenbuffer genannt, da dies relativ gut den Sinn und Zweck des Feldes beschreibt.

```
screenbuffer
.byte $00,$00,$00,$00,$00,$00
```

```

.byte $00,$00,$00,$00,$00,$00

```

Doch mit dem Feld allein ist es nicht getan, wir benötigen noch diverse Unterprogramme, die damit arbeiten.

Folgende drei Unterprogramme müssen wir erstellen:

- **savescrlines:** Sichert die ersten 6 Zeilen des Bildschirms in das Datenfeld scrbuffer
- **clrsclines:** Löscht die ersten 6 Zeilen des Bildschirms, in dem es sie mit Leerzeichen füllt
- **restorescrlines:** Stellt den vorherigen Inhalt der ersten 6 Zeilen des Bildschirms wieder her

Diese drei Unterprogramme sind im folgenden aufgeführt und sollten aufgrund der Kommentare selbsterklärend sein, sodaß es keine umfassenden Erläuterungen von meiner Seite mehr bedarf.

Unterprogramm savescrlines

```

; -----
; savescrlines
; sichert den inhalt der ersten 6
; bildschirmzeilen in das datenfeld
; scrbuffer

; parameter:
; keine

; rueckgabewerte:
; keine

; aendert:
; a,x,status

savescrlines
    ldx #$00

savelineloop
    ; zeichen aus dem
    ; bildschirmspeicher holen
    lda $0400,x

    ; im datenfeld screenbuffer
    ; ablegen
    sta screenbuffer,x

    ; index auf naechstes zeichen
    inx

    ; bereits 240 zeichen gelesen?

```

```

    cpx #$f0
    ; falls nicht, naechstes
    ; zeichen sichern
    bne savelineloop
    ; ansonsten sind wir fertig
    rts

```

Unterprogramm clrsclines

```

;-----+
; clrsclines
; loescht die ersten 6 zeilen des
; bildschirms, diese werden mit
; leerzeichen gefuellt
;
; parameter:
; keine
;
; rueckgabewerte:
; keine
;
; aendert:
; a,x,status
;
clrsclines
    ; fuellzeichen ist leerzeichen
    lda #$20
    idx #$00
;
cirlineloop
    ; leerzeichen schreiben
    sta $0400,x
    ; index auf naechste position
    inx
    ; schon alle 240 positionen
    ; durchlaufen?
    cpx #$f0
    ; falls nicht, weiter zur
    ; naechsten position
    bne cirlineloop
    ; ansonsten sind wir fertig
    rts

```

Unterprogramm restorescrlines

```
;-----  
; restorescrlines  
; stellt die gesicherten ersten 6  
; bildschirmzeilen wieder her  
  
; parameter:  
; keine  
  
; rueckgabewerte:  
; keine  
  
; aendert:  
; a,x,status  
  
restorescrlines  
    ldx #$00  
  
restorelineloop  
    ; zeichen aus datenfeld  
    ; screenbuffer holen  
    lda screenbuffer,x  
    ; in den bildschirmspeicher  
    ; schreiben  
    sta $0400,x  
    ; index auf naechstes zeichen  
    inx  
    ; schon alle 240 zeichen  
    ; wiederhergestellt?  
    cpx #$f0  
    ; falls nicht, naechstes  
    ; zeichen wiederherstellen  
    bne restorelineloop  
    ; ansonsten sind wir fertig  
    rts
```

Nun müssen wir die Aufrufe dieser drei Unterprogramme in der richtigen Reihenfolge kombinieren. Dazu schreiben wir ein Unterprogramm showhelp, welches beim Drücken der Tastenkombination SHIFT + H aufgerufen wird.

Das Unterprogramm ist noch nicht ganz fertig, es fehlt noch die Ausgabe der Hilfetexte, aber der grundsätzliche Ablauf aus Sichern, Löschen und Wiederherstellen der Bildschirmzeilen funktioniert bereits.

```
;-----  
; showhelp  
; zeigt ein fenster mit hilfstexten an  
  
; parameter:  
; keine  
  
; rueckgabewerte:  
; keine  
  
; aendert:  
; a,x,y,status  
  
showhelp
```

```

; die ersten 6 zeilen des
; bildschirms sichern
jsr savescrlines
; zeilen loeschen
jsr clrscrenlines
; auf tastendruck warten

waitkeyloop
    jsr $ffe4
    beq waitkeyloop
; zeilen wiederherstellen
jsr restorescrlines
rts

```

Um die Hilfefunktion auch über die Tastatur verfügbar zu machen, müssen wir wieder das Unterprogramm keyctrl anpassen und die Abfrage der neuen Tastenkombination integrieren.

```

check_clear
    ; sprite loeschen?
    cmp clearkey
    bne check_help
    jsr clrsprite
    jmp keyloop

check_help
    ; fenster mit hilfetexten
    ; anzeigen?
    cmp helpkey
    bne check_quit
    jsr showhelp
    jmp keyloop

```

Dazu ergänzen wir, wie bereits mehrfach bei den anderen Programmfunctionen durchgeführt, einen entsprechenden Abschnitt namens **check_help**, in dem auf die Tastenkombination SHIFT + H reagiert wird, indem das zuvor definierte Unterprogramm **showhelp** aufgerufen wird.

Im Abschnitt **check_clear** mussten wir auch das Sprungziel nach dem Befehl BNE von **check_quit** auf **check_help** abändern, damit die neue Tastenkombination in die Kette der Vergleichsabfragen integriert wird.

Den aktuellen Stand des Programms finden Sie unter dem Namen **SAVESCR** auf der Diskette. Wenn Sie die Tastenkombination SHIFT + H drücken, werden die ersten 6 Zeilen des Bildschirms gelöscht und nach einem beliebigen Tastendruck ihr ursprünglicher Inhalt wiederhergestellt.

Nun müssen wir nur noch den Rahmen um das Fenster bzw. die Hilfsinformationen darin anzeigen und wir haben unseren Sprite-Editor fertiggestellt!

Dazu definieren wir uns im Datenabschnitt zunächst die Texte, die wir in dem Fenster anzeigen wollen.

```

helpsetreset
    .text "set/reset bit: "
    .null "return/space"

helpsaveload
    .text "save/load: shift + s"
    .null " / shift + l"

helpclear
    .text "clear: shift + clear/"
    .null "home"

helpquit
    .null "quit: shift + q"

```

Für das Befüllen des Hilfenummers habe ich ebenfalls ein eigenes Unterprogramm namens drawhelpwindow geschrieben. Das wäre zwar nicht unbedingt nötig gewesen, aber ich finde, es sorgt für ein wenig mehr Übersichtlichkeit.

```

;-----  

; drawhelpwindow  

; zeichnet einen rahmen um die ersten  

; 6 bildschirmzeilen und gibt darin  

; hilfsinformationen aus  

;  

; parameter:  

; keine  

;  

; rueckgabewerte:  

; keine  

;  

; aendert:  

; a,x,y,status  

;  

drawhelpwindow  

    ; ecke links oben  

    lda #$55  

    sta $0400  

    ; ecke rechts oben  

    lda #$49  

    sta $0427  

    ; ecke links unten  

    lda #$4a  

    sta $04c8  

    ; ecke rechts unten  

    lda #$4b  

    sta $04ef  

    ; horizontale rahmenlinien  

    lda #$40  

    ldx #$00  

;  

horizloop  

    sta $0401,x  

    sta $04c9,x  

    inx  

    cpx #$26  

    bne horizloop  

    ; vertikale rahmenlinien  

    lda #$42  

    sta $0428  

    sta $044f

```

```

sta $0450
sta $0477
sta $0478
sta $049f
sta $04a0
sta $04c7

; hilfetexte ausgeben

idx #$01
idy #$06
lda #<helpsetreset
sta $fd
lda #>helpsetreset
sta $fe
jsr printstr

idx #$02
idy #$04
lda #<helpsaveload
sta $fd
lda #>helpsaveload
sta $fe
jsr printstr

idx #$03
idy #$05
lda #<helpclear
sta $fd
lda #>helpclear
sta $fe
jsr printstr

idx #$04
idy #$0b
lda #<helpquit
sta $fd
lda #>helpquit
sta $fe
jsr printstr

rts

```

Hier werden zuerst die abgerundeten Ecken gezeichnet. Dazu wird einfach der Screencode des jeweiligen Zeichens in den Akkumulator geschrieben und an die entsprechende Adresse im Bildschirmspeicher geschrieben.

Als nächstes sind die horizontalen Linien am oberen und unteren Rand des Rahmens an der Reihe.

Dies wird über eine Schleife realisiert, in der abwechselnd ein Teilstrich am oberen und unteren Rand in den Bildschirmspeicher geschrieben wird.

Die Teile der vertikalen Linien habe ich einzeln in den Bildschirmspeicher geschrieben, da es nicht soviele sind und eine Schleife keinen Mehrwert gebracht hätte.

Somit ist der Rahmen schon mal gezeichnet.

Nun müssen wir nur noch die Hilfetexte in dem Fenster ausgeben.

Dies wird über das bereits bekannte Unterprogramm printstr realisiert, das wir ganz zu Beginn unserer Arbeit erstellt haben.

Als Parameter werden die gewünschte Position des Strings im X Register und Y Register sowie die Stringadresse aufgeteilt auf die Speicherstellen \$FD und \$FE übergeben.

Dieses Schema wenden wir auf alle Strings an, die wir uns vorhin für die Hilfetexte definiert haben.

Nun müssen wir im Unterprogramm showhelp nur noch den Aufruf des Unterprogramms drawhelpwindow einbauen und wir haben auch die Hilfefunktion erfolgreich umgesetzt.

```
; zeilen loeschen
jsr clrscrlns
; hilfsinformationen anzeigen
jsr drawhelpwindow
```

Die finale Fassung des Spriteeditors habe ich unter dem Namen SPRITEEDITOR auf der Diskette zur Verfügung gestellt.

So, nun haben wir es endlich geschafft und unser Projekt Sprite-Editor erfolgreich umgesetzt!

Wie eingangs erwähnt, habe ich ihn so programmiert, dass noch Platz für Verbesserungen bleibt, die Sie als Übung nach Ihren eigenen Vorstellungen umsetzen können wenn sie gerne möchten. Ich hoffe, dass ich Ihnen mit diesem Buch einen erfolgreichen Einstieg in die Assembler-Programmierung auf dem Commodore 64 ermöglichen konnte.

Wenn Sie mit den erworbenen Fähigkeiten nun Ihre eigenen Assembler-Programme erstellen können, dann habe ich mein Ziel erreicht :)

Erinnern Sie sich noch an unser erstes Maschinen-Programm? Wir haben eine Zahl in das X Register geschrieben und sind dann gleich wieder zu Basic zurückgekehrt. Das Programm bestand also aus ganzen 3 Bytes und nun sehen Sie sich an, wo wir heute stehen!

Wir haben ein ausgewachsenes Assembler-Programm geschrieben, das nicht nur einen praktischen Nutzen hat, sondern Ihnen (so hoffe ich) auch einen erfolgreichen Einstieg in diese Art der Programmierung verschafft hat.