Einführung in Maschinensprache und Assembler auf dem C64

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, denn dies ist die einzige Sprache, die die CPU des C64 letztendlich versteht.

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.

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.

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 Register direkt auf der CPU und den 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, soviele wie es CPU-Typen gibt.

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

Anmerkung:

Da nachfolgend sehr oft Zahlenwerte aus unterschiedlichen Zahlensystemen angegeben werden, werde ich zur Unterscheidung hexadezimale Werte mit einem vorangestellten $, binäre Werte mit einem vorangestellten % und dezimale Werte ohne vorangestelltes Zeichen angeben.

Eine Ausnahme werde ich bei Sequenzen aus Befehlscodes machen. Dort werde ich das $ vor den hexadezimalen Werten weglassen, aber explizit darauf hinweisen, dass es sich um hexadezimale Werte handelt.

Die CPU des C64 verfügt beispielsweise über einen Befehl, mit dem man eine Zahl in das oben genannte X Register schreiben kann. Dieser Befehl hat den Befehlscode 162, das entspricht $A2 in hexadezimaler bzw. %10100010 in binärer Schreibweise.

Zu den Befehlscodes eine wichtige Anmerkung:

Sie müssen sich diese Zahlen nicht merken, 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.

Will man nun der CPU diesen Befehl erteilen, muss man diesen Wert 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 BASIC 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 aus Basic heraus 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.

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.

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.

Wir können das Programm das wir soeben in den Speicher geschrieben haben, sogar ausführen, denn der Basic Befehl SYS ist genau für diesen Zweck gedacht.

Unser Programm beginnt an der Adresse 5376, 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, welcher Wert nach Ausführung des Programms im X Register steht.

Wenn wir ein Maschinenprogramm mit SYS gestartet haben, kann man nachdem das Programm durchgelaufen ist, den Inhalt des X Registers über die Speicherstelle 781 abfragen.
Wie im folgenden Screenshot zu sehen, liefert PEEK (781) das korrekte Ergebnis 200 zurück.

Es gibt übrigens auch noch für andere Register eine solche Speicherstelle, hier eine Zusammenfassung:

Register	**Speicherstelle**
Akkumulator	780
X Register	781
Y Register	782
Statusregister	783

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, 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. 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 lässt sich nicht durch ein Byte allein darstellen, d.h. wir brauchen zwei Bytes dafür.

Die CPU liest die beiden Bytes von 16 Bit Zahlen in umgekehrter Reihenfolge aus dem Arbeitsspeicher, das heißt wir müssen zuerst das niederwertige Byte (also die ersten 8 Bits von rechts) und dann das höherwertige Byte in den Speicher schreiben damit die richtige Zahl eingelesen wird.

Wenn man 1024 ins Hexadezimalsystem umrechnet ergibt das $0400, das niederwertige Byte enthält also dezimal 0 und das höherwertige Byte dezimal 4.

Gehen wir nun unser Programm Byte für Byte durch.

Wir lassen das Programm wieder an der Adresse 5376 beginnen.

Adresse	**Inhalt (dezimal)**	**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)

Wir schreiben nun also mit POKE-Befehlen unser Programm Byte für Byte in den Speicher:

```
10 POKE 5376,162
20 POKE 5377,4
30 POKE 5378,142
40 POKE 5379,0
50 POKE 5380,4
60 POKE 5381,96
```

Speichern Sie das BASIC-Programm mit SAVE „SHOWD“,8

Starten Sie es mit RUN, damit unser Maschinenprogramm in den Speicher geschrieben wird.

Und starten selbiges mit dem Befehl:

SYS 5376

Es müsste dann 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.

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

Das Programm soll 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 beim 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 enthält also dezimal 210 und das höherwertige Byte 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, diesesmal 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.

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 erniedrigen
- 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 diesesmal etwas mehr Zeilen haben, werden wir die POKE-Anweisungen in ein 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
```

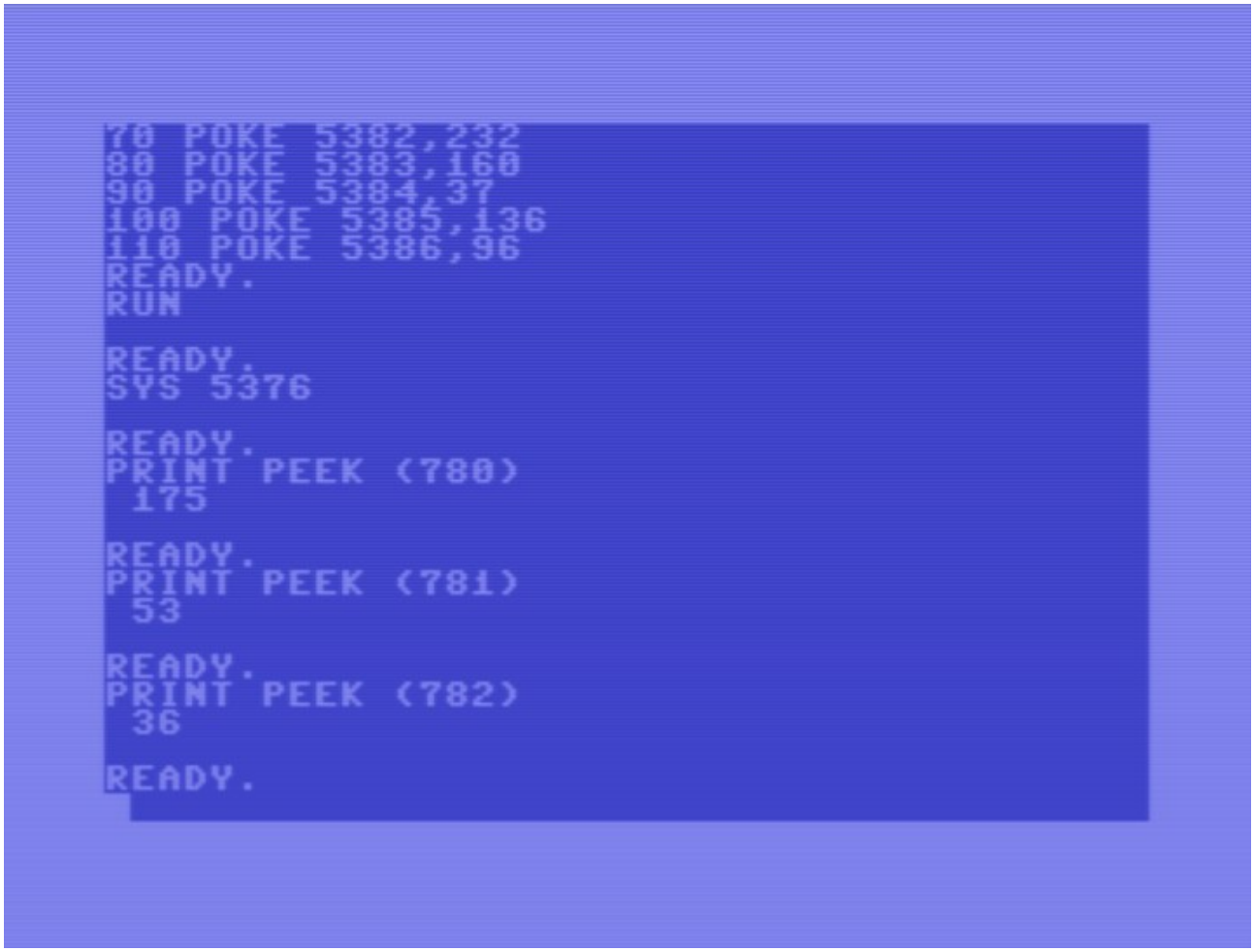
Speichern Sie dieses Programm mit SAVE „CALC1“,8 auf Diskette ab.

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.



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!

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

Um den Wert des Akkumulators in eine Speicherstelle zu schreiben, brauchen wir den Maschinenbefehl mit dem Befehlscode 141.

Der Befehlscode um den Inhalt des X Registers in eine Speicherstelle zu schreiben, lautet 142 und jener um den Inhalt des Y Registers in eine Speicherstelle zu schreiben lautet 140.

Diese Werte müssen wir in drei Speicherstellen schreiben, deren Adressen wir uns erst ausrechnen müssen.

Eines wissen wir schon jetzt: Es handelt sich um 16 Bit Adressen, da unser Programm ja bereits an der Adresse 5376 beginnt.

Um diese drei 16 Bit Adressen im Maschinenprogramm anzugeben, brauchen wir also 3 x 2 Bytes, also 6 Bytes.

Dazu brauchen wir noch 3 Bytes für die drei Befehle, welche den Inhalt des Akkumulators, des X Registers und des Y Registers in diese Speicherstellen schreiben.

Wir brauchen also zusätzliche 9 Bytes in unserem Programm.

Als ersten Schritt löschen wir folgende Zeile in unserem Basic Programm:

110 POKE 5386,96

Dann fügen wir 9 neue Zeilen mit POKE-Befehlen ein, lassen aber den Wert selbst noch frei.

Wir legen also folgende 9 zusätzliche Zeilen an:

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

Und fügen als letzte Zeile diese hier an:

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 im BASIC Programm die Befehlscodes sowie die niederwertigen und höherwertigen Bytes der Adressen eintragen:

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

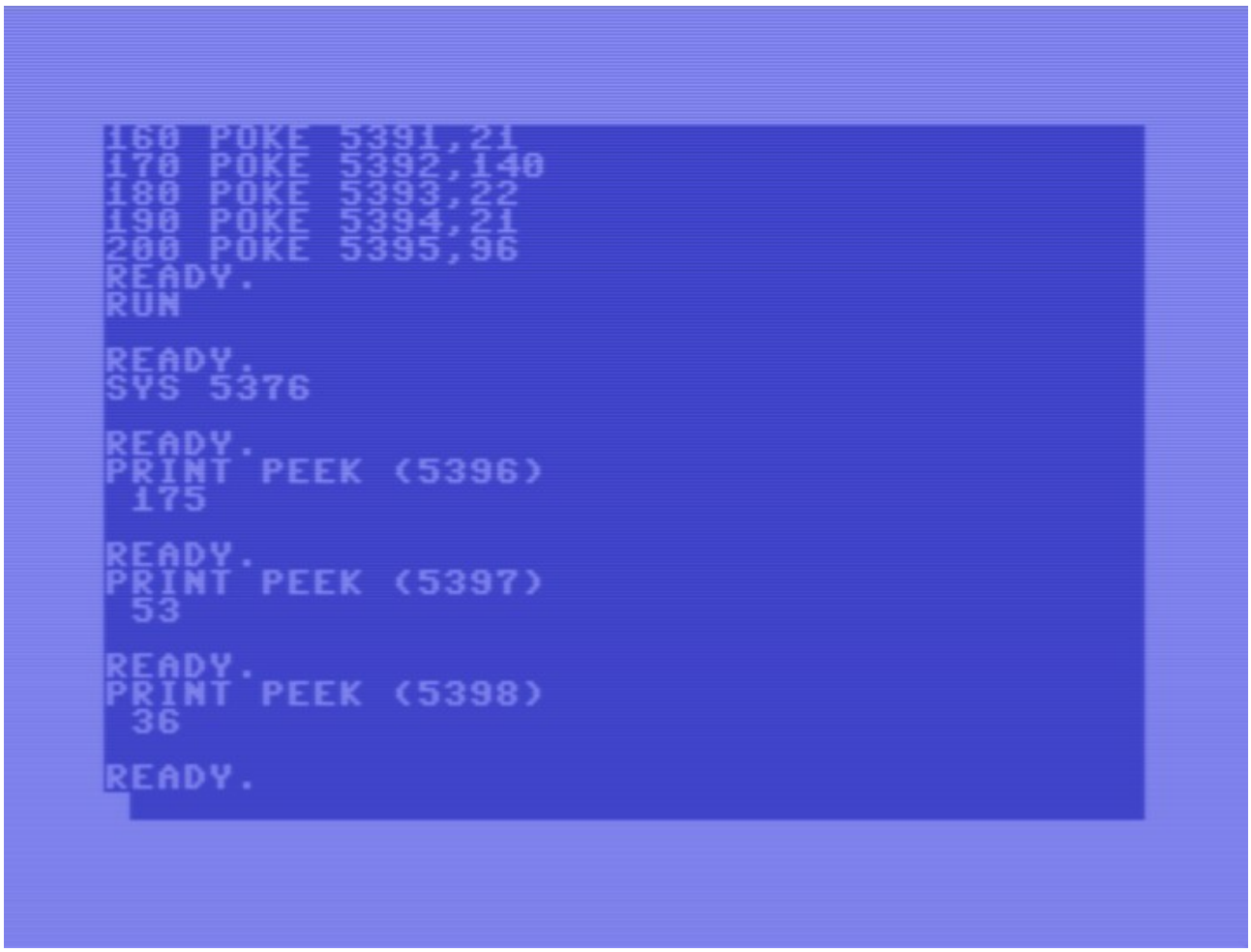
Adresse	Inhalt	Beschreibung
5387	20	Niederwertiges Byte der Adresse 5396
5388	21	Höherwertiges Byte der Adresse 5396
5389	142	Befehlscode zum Speichern des Inhalts des X Registers in einer Speicherstelle
5390	21	Niederwertiges Byte der Adresse 5397
5391	21	Höherwertiges Byte der Adresse 5397
5392	140	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!



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 :)

Anmerkung:

Dass ich die bisherigen Maschinenprogramme mit POKE einzeln Byte für Byte in den Speicher geschrieben habe, diente nur der besseren Veranschaulichung.

Normalerweise verwendet man für das Übertragen eines Maschinenprogramms in den Arbeitsspeicher ein kleines BASIC-Programm, welches die Zahlencodes in Form von DATA-Zeilen enthält und diese per FOR-Schleife in den Speicher schreibt.

Hier zum Vergleich die obige Vorgangsweise, bei der ich die Werte einzeln in den Speicher geschrieben habe:

```
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 5386,141
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
200 POKE 5395,96
```

Und hier die Methode über das BASIC-Programm:

```
10 FOR I=5376 to 5395
20 READ A
30 POKE I,A
40 NEXT I
50 END
60 DATA 169, 100, 105, 75, 162, 52, 232, 160, 37, 136, 141, 20, 21, 142, 21, 21, 140, 22, 21, 96
```

Dieses Programm wird mit RUN gestartet und nach Beendigung steht das Maschinenprogramm im Speicher.

Dieses BASIC-Programm kann dann mit NEW gelöscht werden, da es nicht mehr benötigt wird.

Das Maschinenprogramm kann man dann mit SYS + Startadresse starten, hier wie bereits durchgeführt mit SYS 5376.

So, das war's für's Erste mit dem mühsamen Programmieren direkt in Maschinensprache. Die bisherigen und teilweise mühsamen Darstellungen waren jedoch unverzichtbar für das Verständnis, wie die CPU ein Programm verarbeitet und ich hoffe, dass ich alle Abläufe gut verständlich erklären konnte.

Weiter geht es nun endlich mit der Programmierung in Assembler, der Komfort-Version der Maschinensprache :)

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 Zahlen übersetzt werden.

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.

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. Die CPU hat da 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.

Betrachten wir beispielsweise einen der Maschinenbefehle, mit dem wir bisher oft gearbeitet haben, den Maschinenbefehl zum Laden des X Registers.

Will man in Maschinensprache das X Register mit dem Zahlenwert 200 laden, so lautet die zugehörige Zahlenfolge 162 200.

In Assemblersprache formuliert würde das so aussehen:

LDX #200

Das sieht doch schon viel verständlicher aus oder? LDX soll "Load X" bedeuten. Was es mit dem Zeichen # auf sich hat, erkläre ich in Kürze.

Oder nehmen wir den Maschinenbefehl, mit dem wir den Zeichencode des Buchstaben D in die Speicherstelle 1024 geschrieben haben:

Maschinensprache: 142 0 4
Assembler: STX 1024

STX soll "Store X" bedeuten.

Dann hatten wir noch den Maschinenbefehl, welcher das Ende unseres Maschinenprogramms markiert hat:

Maschinensprache: 96
Assembler: RTS

RTS soll "Return To Subroutine" bedeuten.

Nachfolgend eine Gegenüberstellung des Gesamtprogramms:

Maschinensprache: 162, 4, 142, 0, 4, 96

Assembler:

LDX #4
STX 1024
RTS

Das liest sich doch im Unterschied zu vorher ja quasi wie von selbst oder? Naja, nicht ganz, aber es ist zumindest schon mal eine enorme Verbesserung was die Lesbarkeit des Programms betrifft.

LDX #4   => Lade das X Register mit dem Wert 4
STX 1024 => Speichere den Inhalt des X Registers in der Speicherstelle 1024
RTS   => Ende des Programms, kehre zum Aufrufer zurück

Aber was hat es nun mit dem Zeichen # auf sich?

Dadurch wird gekennzeichnet, dass hier der Zahlenwert 4 gemeint ist und nicht etwa die Speicheradresse Nummer 4.

Wenn wir tatsächlich die Speicherstelle Nummer 4 meinen, dann lassen wir das # einfach weg.

Womit wir bei einem weiteren Vorteil der Assemblersprache sind:

Man muss sich keine Gedanken mehr um die (richtigen) Befehlscodes machen, denn bei der Übersetzung erzeugt der Assembler (nun ist der Übersetzer gemeint) automatisch den richtigen Befehlscode. Er erkennt anhand von speziellen Markierungen (z.B. dem #) was gemeint ist.

Übersetzen wir nun mal zur Veranschaulichung die drei Varianten des Maschinenbefehls zum Laden des X Registers, die wir vorhin kennengelernt haben.

Mensch	**Maschinensprache**	**Assembler**	**Anmerkung**
Lade das X Register mit dem Wert 123	162 123	LDX #123	Anhand der # erkennt der Assembler dass hier der Zahlenwert 123 gemeint ist und generiert die Zahlenfolge 162 123
Lade das X Register mit dem Inhalt der Speicherstelle 1024 (Adresse ist größer als 255)	174 0 4	LDX 1024	Da die Nummer der Speicherstelle größer als 255 ist generiert der Assembler die Zahlenfolge 174 0 4
Lade das X Register mit dem Inhalt der Speicherstelle 200 (Adresse liegt zwischen 0 und 255)	166 200	LDX 200	Da die Nummer der Speicherstelle zwischen 0 und 255 liegt, sich also mit einem einzigen Byte darstellen lässt, generiert der Assembler die Zahlenfolge 166 200

Wir sehen hier immer denselben Assembler-Befehl LDX, aber durch die unterschiedliche Art der Parameter wird unterschiedlicher Maschinencode generiert, wie in der zweiten Spalte zu sehen ist.

Als weitere Demonstration der besseren Lesbarkeit von Assembler-Programmen hier auch die Assembler-Variante unseres letzten Maschinensprache-Programms, welches einige Rechenoperationen durchgeführt hat:

Befehl	**Beschreibung**
LDA #100	Akkumulator mit Wert 100 laden
ADC #75	Den Wert 75 zum Inhalt des Akkumulators addieren
LDX #52	X Register mit Wert 52 laden
INX	Inhalt des X Registers um 1 erhöhen
LDY #37	Y Register mit Wert 37 laden
DEY	Inhalt des Y Registers um 1 erniedrigen
STA 5396	Inhalt des Akkumulators an die Speicherstelle 5396 schreiben

Befehl	Beschreibung
STX 5397	Inhalt des X Registers an die Speicherstelle 5397 schreiben
STY 5398	Inhalt des Y Registers an die Speicherstelle 5398 schreiben
RTS	Programm beenden und zurück zu BASIC

Das lässt sich doch um einiges einfacher lesen als die Zahlenfolge

169, 100, 105, 75, 162, 52, 232, 160, 37, 136, 141, 20, 192, 142, 21, 192, 140, 22, 192, 96

oder?

Man sieht also bereits, Assemblerprogramme sind nicht nur deutlich lesbarer als Maschinenprogramme, sondern man bleibt auch von den Befehlscodes verschont, von denen es ja, wie wir gesehen haben, mehrere für ein und denselben Befehl gibt, je nachdem welche Parameter wir angeben wollen.

Anmerkung:

Zu den Befehlen INX und DEY gibt es übrigens auch die Gegenstücke DEX (vermindern des Inhalts des X Registers um 1) und INY (erhöhen des Inhalts des Y Registers um 1)

Ok, aber nun stellt sich die große Frage:

Wie programmiert man in Assembler und wo kriegt man den Assembler (also das Übersetzungsprogramm) her?

Es gibt mehrere Assembler für den C64, z.B. den SMON, den Turbo Assembler, den Hypra-Ass und noch einge andere.

Für die Programmierung werden wir zunächst den SMON verwenden, da er einfach zu verwenden und für unsere Zwecke mehr als ausreichend ist. Später, wenn unsere Programme um einiges komplexer werden, steigen wir auf den wesentlich komfortableren Turbo Macro Pro um.

Programmierung mit SMON

Bevor wir beginnen, fassen wir nochmal in einer Tabelle zusammen, welche Maschinenbefehle wir bis jetzt kennengelernt haben. Diese Tabelle werden Sie in Kürze brauchen wenn wir beginnen, mit SMON zu arbeiten.

Bei den Befehlen zum Laden des Akkumulators, des X Registers und des Y Registers habe ich zum Vergleichen folgende Varianten aufgeführt, die sich ja wie erwähnt durch den Befehlscode und die Art der Parameter unterscheiden.

- Laden des jeweiligen Registers mit einem Zahlenwert, ich habe hier in der Tabelle immer den Zahlenwert 200 (hexadezimal $C8) verwendet.

- Laden des jeweiligen Registers mit dem Inhalt einer Speicherstelle, deren Adresse größer als 255 ist, also einer 16 Bit Adresse, welche durch 2 Bytes dargestellt wird.
  Ich habe hier in der Tabelle immer die Adresse 1024 (hexadezimal $0400) verwendet.

- Laden des jeweiligen Registers mit dem Inhalt einer Speicherstelle, deren Adresse zwischen 0 und 255 liegt, also einer 8 Bit Adresse, für deren Darstellung ein einziges Byte ausreicht.
  Ich habe hier in der Tabelle immer die Adresse 100 (hexadezimal $64) verwendet.

Bei den Befehlen zum Übertragen des Inhalts des Akkumulators, des X Registers und des Y Registers in eine Speicherstelle im Arbeitsspeicher, habe ich zum Vergleichen folgende Varianten aufgeführt:

- Übertragen des jeweiligen Registerinhalts in eine Speicherstelle, deren Adresse größer als 255 ist, also einer 16 Bit Adresse, welche durch 2 Bytes dargestellt wird.
  Ich habe hier in der Tabelle immer die Adresse 1024 (hexadezimal $0400) verwendet.

- Übertragen des jeweiligen Registerinhalts in eine Speicherstelle, deren Adresse zwischen 0 und 255 liegt, also einer 8 Bit Adresse, für deren Darstellung ein einziges Byte ausreicht.
  Ich habe hier in der Tabelle immer die Adresse 100 (hexadezimal $64) verwendet.

Befehl	**Befehlscode**	**Parameter**	**Assembler-Befehl**	**Maschinensprache (dezimal)**	**Maschinensprache (hexadezimal)**
Laden des Akkumulators mit einem Zahlenwert	169 ($A9)	Zahlenwert 200 ($C8)	LDA #$C8	169 200	A9 C8
Laden des Akkumulators mit dem Inhalt einer Speicherstelle mit Adresse > 255	173 ($AD)	Adresse > 255 Adresse 1024 ($0400)	LDA $0400	173 0 4	AD 00 04
Laden des	165 ($A5)	Adresse zwischen	LDA $64	165 100	A5 64

Befehl	**Befehlscode**	**Parameter**	**Assembler-Befehl**	**Maschinensprache (dezimal)**	**Maschinensprache (hexadezimal)**
Akkumulators mit einer Speicherstelle mit Adresse zwischen 0 und 255		0 und 255 Adresse 100 ($64)			
Laden des X Registers mit einem Zahlenwert	162 ($A2)	Zahlenwert 200 ($C8)	LDX #$C8	162 200	A2 C8
Laden des X Registers mit dem Inhalt einer Speicherstelle mit Adresse > 255	174 ($AE)	Adresse > 255 Adresse 1024 ($0400)	LDX $0400	174 0 4	AE 00 04
Laden des X Registers mit dem Inhalt einer Speicherstelle mit Adresse zwischen 0 und 255	166 ($A6)	Adresse zwischen 0 und 255 Adresse 100 ($64)	LDX $64	166 100	A6 64
Laden des Y Registers mit einem Zahlenwert	160 ($A0)	Zahlenwert 200 ($C8)	LDY #$C8	160 200	A0 C8
Laden des Y Registers mit dem Inhalt einer Speicherstelle mit Adresse > 255	172 ($AC)	Adresse > 255 Adresse 1024 ($0400)	LDY $0400	172 0 4	AC 00 04
Laden des Y Registers mit dem Inhalt einer Speicherstelle mit Adresse zwischen	164 ($A4)	Adresse zwischen 0 und 255 Adresse 100 ($64)	LDY $64	164 100	A4 64

Befehl	**Befehlscode**	**Parameter**	**Assembler-Befehl**	**Maschinensprache (dezimal)**	**Maschinensprache (hexadezimal)**
0 und 255					
Übertragen des Inhalt des Akkumulators an eine Speicherstelle mit Adresse > 255	141 ($8D)	Adresse > 255 Adresse 1024 ($0400)	STA $0400	141 0 4	8D 00 04
	133 ($85)	Adresse zwischen 0 und 255 Adresse 100 (64)	STA $64	133 100	85 64
Übertragen des Inhalt des X Registers an eine Speicherstelle mit Adresse > 255	142 ($8E)	Adresse > 255 Adresse 1024 (0400)	STX $0400	142 0 4	8E 00 04
Übertragen des Inhalt des X Registers an eine Speicherstelle mit Adresse zwischen 0 und 255	134 ($86)	Adresse zwischen 0 und 255 Adresse 100 ($64)	STX $64	134 100	86 64
Übertragen des Inhalt des Y Registers an eine Speicherstelle mit Adresse > 255	140 ($8C)	Adresse > 255 Adresse 1024 ($0400)	STY $0400	140 0 4	8C 00 04
Übertragen des Inhalt des Y Registers an eine Speicherstelle mit Adresse zwischen 0 und 255	132 ($84)	Adresse zwischen 0 und 255 Adresse 100 ($64)	STY $64	132 100	84 64
Erhöhen des Inhalts des	232 ($E8)		INX	232	E8

Befehl	Befehlscode	Parameter	Assembler-Befehl	Maschinensprache (dezimal)	Maschinensprache (hexadezimal)
X Registers um 1					
Erhöhen des Inhalts des Y Registers um 1	200 ($C8)		INY	200	C8
Vermindern des Inhalts des X Registers um 1	202 ($CA)		DEX	202	CA
Vermindern des Inhalts des Y Registers um 1	136 ($88)		DEY	136	88
Zurück zu BASIC	96 ($60)		RTS	96	60

Was ist SMON und was kann man damit alles machen?

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.

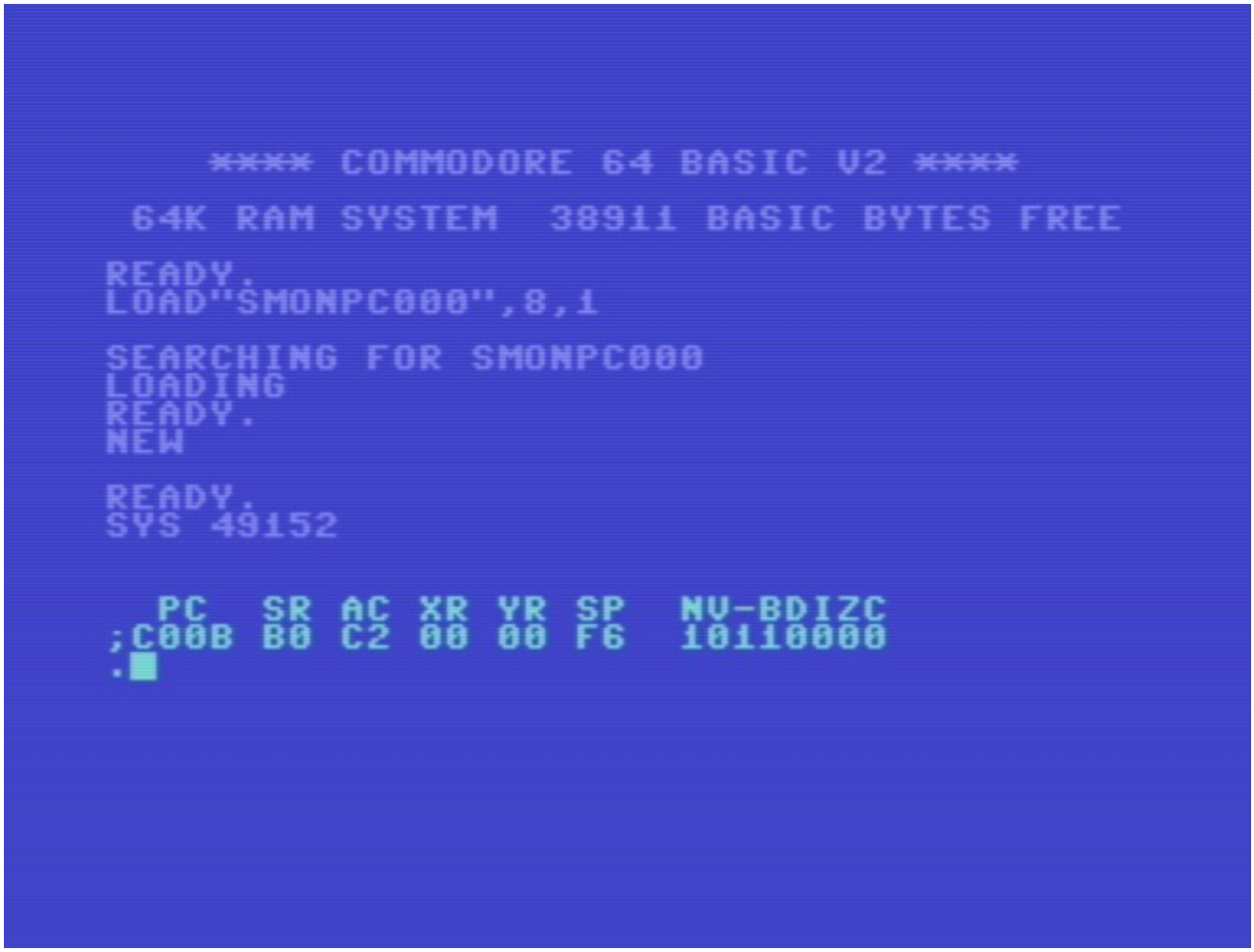
Auf der Webseite https://www.c64-wiki.de/wiki/SMON ist eine sehr umfassende Beschreibung des SMON zu finden die Sie sicher immer wieder mal gut gebrauchen können.

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.

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.



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 „#“ gefolgt von einer dezimalen Zahl eingeben liefert uns SMON die entsprechende hexadezimale Darstellung.

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

Liefert uns das Ergebnis:

```
   PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F6  10110000
i1500  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 von „$“ gefolgt von der hexadezimalen Zahl.

```
   PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F6  10110000
i1500  5376
.$1500
```

Liefert uns das Ergebnis:

```
   PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F6  10110000
i1500  5376
i1500  5376
.
```

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

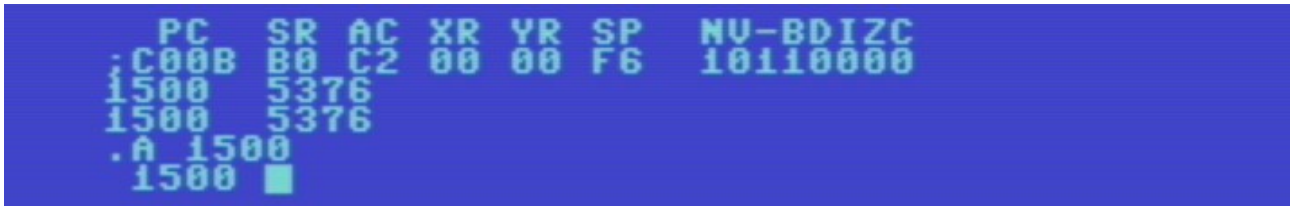
Damit wir nun mit der Eingabe ab Adresse $1500 beginnen können, müssen wir dem SMON dem Befehl „A“ gefolgt von der Adresse $1500 geben.

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

Geben wir also folgendes ein:

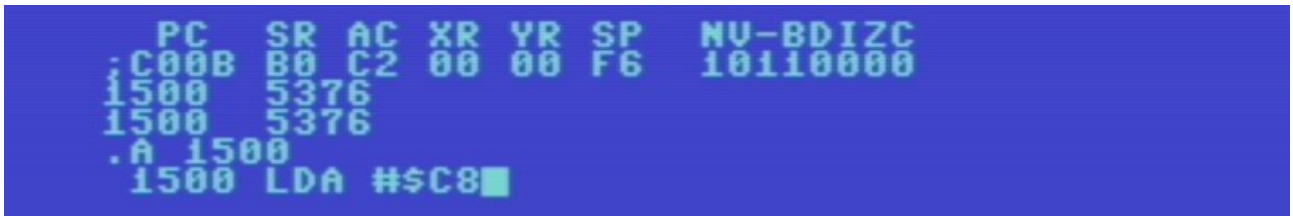
```
   PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F6  10110000
i1500  5376
i1500  5376
.A 1500
```

SMON meldet sich mit folgender Anzeige:



Nun können wir unseren ersten Assembler-Befehl eingeben.

Nehmen wir den ersten Befehl in unserer Liste oben welcher LDA #$C8 lautet.



Und bekommen folgende Rückmeldung:

Was ist passiert? SMON hat unseren Befehl LDA #$C8 entgegengenommen und verarbeitet.

Wie ist er dabei vorgegangen?

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.

$C8 steht hier für die hexadezimale Form der dezimalen Zahl 200.

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 $C8, der Befehl beansprucht also zwei Bytes.

$A9 ist die hexadezimale Form des dezimalen Befehlscodes 169 und $C8 wie gesagt die hexadezimale Darstellung des dezimalen Wertes 200.

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 aus unserer Tabelle ein (LDA $0400)

Und erhalten folgendes Ergebnis:

Diesesmal haben wir das Zeichen „#“ hinter dem Befehl LDA weggelassen, um dem SMON anzuzeigen, dass wir nun keinen Zahlenwert sondern eine Speicheradresse meinen.

Das Zeichen „$“ kennzeichnet wiederum eine hexadezimale Zahl, in diesem Fall $0400, welche die hexadezimale Form der dezimalen Zahl 1024 darstellt.

Das höherwertige Byte lautet $04 und das niederwertige Byte $00.

Angenehm ist hier, dass man diese beiden Bytes nicht in umgekehrter Reihenfolge eingeben muss, wie wir es beim Programmieren in Maschinensprache tun mussten.

Der SMON sorgt bei der Übersetzung in Maschinensprache eigenständig dafür, dass die Reihenfolge stimmt.

Und tatsächlich, wenn wir uns die Zahlenfolge links neben dem Befehl LDA ansehen, sieht man, dass zuerst das Byte $00 und erst dann das Byte $04 in den Speicher geschrieben wird.

In der Speicherstelle $1502 steht nun der Befehlscode $AD, in der Speicherstelle $1503 das niederwertige Byte $00 und in der Speicherstelle $1504 das höherwertige Byte $04.

Der Befehl nimmt nun im Unterschied zum vorherigen Befehl 3 Bytes Speicher in Anspruch ($AD für den Befehlscode und zwei weitere Bytes für das niederwertige und höherwertige Byte der Speicheradresse $0400)

Sehen Sie den Unterschied in den Befehlscodes? In der ersten Zeile sieht man den Befehlscode $A9 und in der zweiten Zeile den Befehlscode $AD, obwohl wir in beiden Fällen den Befehl LDA verwendet haben.

Das ergibt sich aus der unterschiedlichen Angabe, die wir beim Parameter gemacht haben.

#$C8 => wir meinen den Zahlenwert $C8
$0400 => wir meinen die Speicheradresse $0400

Weiter geht's mit dem nächsten Befehl aus der Tabelle, welcher an der Speicheradresse $1505 landet.

Wir geben ein: LDA $64

Mit folgendem Ergebnis:

Bei diesem LDA-Befehl haben wir wiederum kein Zeichen „#“ angegeben, d.h. wir meinen eine Speicheradresse.

Der Unterschied zum vorherigen Befehl ist jedoch der, dass wir diesesmal die hexadezimale Adresse $64 angegeben haben, womit die Speicherstelle mit der dezimalen Adresse 100 gemeint ist.

Da dieser Wert zwischen 0 und 255 liegt, kommen wir mit einem einzigen Byte aus.

Auch hier sehen wir wieder einen Unterschied im Befehlscode, denn wir haben wie bisher den Befehl LDA verwendet und eine Speicheradresse als Parameter angegeben, doch der SMON erkennt, dass der Wert sich mit einem einzigen Byte darstellen lässt und generiert dementsprechend nicht wie vorhin den Befehlscode $AD, sondern den Befehlscode $A5.

Dadurch „weiß“ die CPU:

Nach dem Befehlscode folgt eine Speicheradresse, aber diesesmal ist sie nur ein Byte anstatt zwei Bytes lang.

Der Befehl nimmt im Unterschied zum vorherigen Befehl also nur zwei Bytes in Anspruch, wie an den Zahlen neben dem Befehl LDA zu sehen ist (A5 64)

Geben Sie doch mal zur Übung die restlichen Befehle aus der Tabelle ein und vergleichen Sie die Ergebnisse mit den Angaben in der Tabelle bzw. prüfen Sie, ob Sie nachvollziehen können, warum die generierten Zahlenfolgen so aussehen und nicht anders.

Anmerkung: Geben Sie jedoch anstelle des Befehls RTS diesesmal den Befehl BRK ein.

Wenn Sie alle Befehle aus der Tabelle eingegeben haben, sollte die Anzeige folgendermaßen aussehen:

Der Grund, warum wir hier nicht den Befehl RTS sondern den Befehl BRK mit dem Befehlscode 0 verwenden, ist der, dass dieser bewirken würde, dass SMON verlassen wird und wir wieder in der BASIC-Umgebung landen.

Durch den Befehl BRK zeigen wir ebenfalls das Ende unseres Programms an, doch wir verbleiben im SMON, weil wir ja schließlich noch vieles ausprobieren wollen.

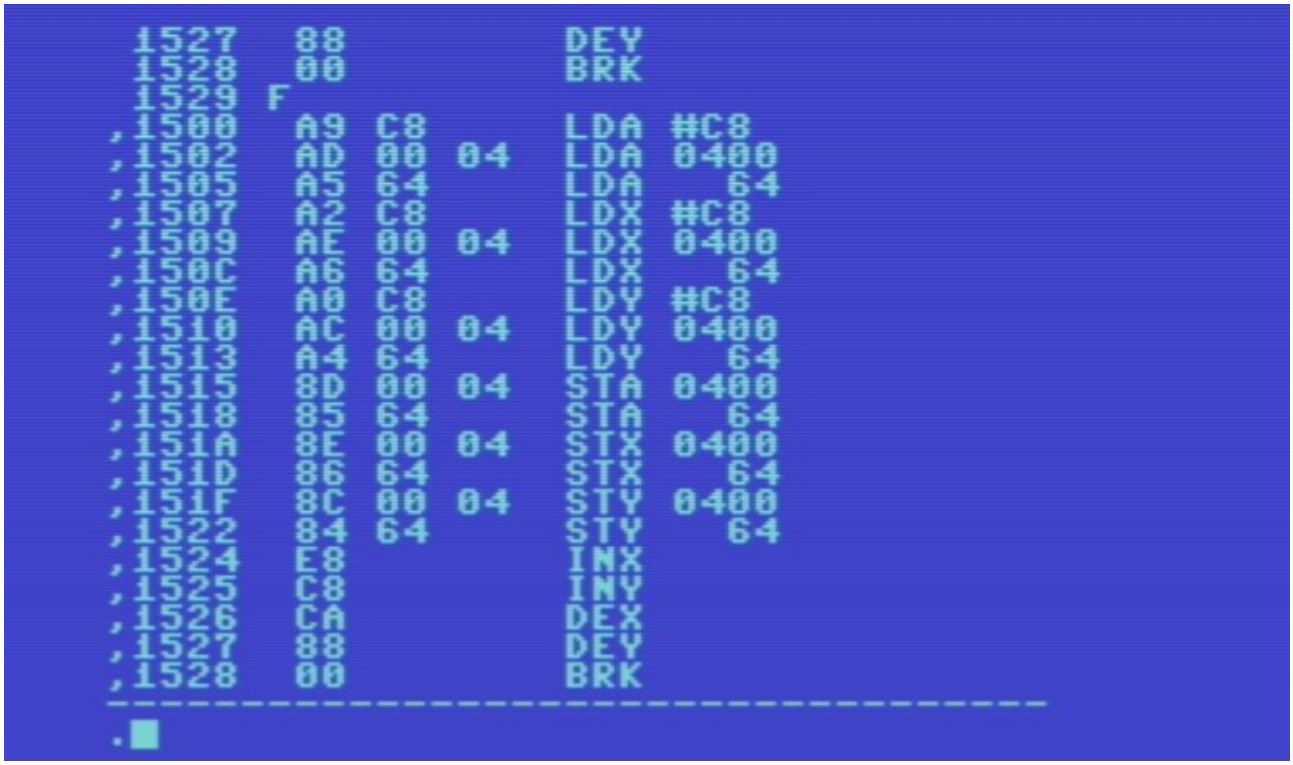
Sollten Sie jedoch versehentlich wieder in der BASIC-Umgebung gelandet sein, können Sie jederzeit wieder durch Eingabe von

SYS 49152

zu SMON zurückkehren und weiterarbeiten. Das Programm, das Sie eingegeben haben, wurde durch den Wechsel zu BASIC nicht gelöscht, es ist immer noch im Arbeitsspeicher ab der Adresse $1500 vorhanden.

Nun wollen wir die Programmeingabe verlassen was mittels Eingabe des SMON-Befehls „F“ wie Finish möglich ist.

Die nächste Anzeige sollte dann so aussehen:



SMON verlässt den Programmeingabe-Modus, gibt uns unser Programm nochmals aus und wir landen wieder im Befehlseingabe-Modus. Die Beistriche am Anfang neben den Zeilen bedeuten, dass diese Zeilen durch uns veränderbar sind, wie wir bald sehen werden.

Unser Programm beginnt bei Adresse $1500 und endet bei Adresse $1528 mit dem BRK Befehl.

Das Programm macht an sich nichts Sinnvolles, es sollte viel mehr dazu dienen Ihnen zu zeigen, wie man überhaupt ein Assembler-Programm eingibt und was noch viel wichtiger ist:

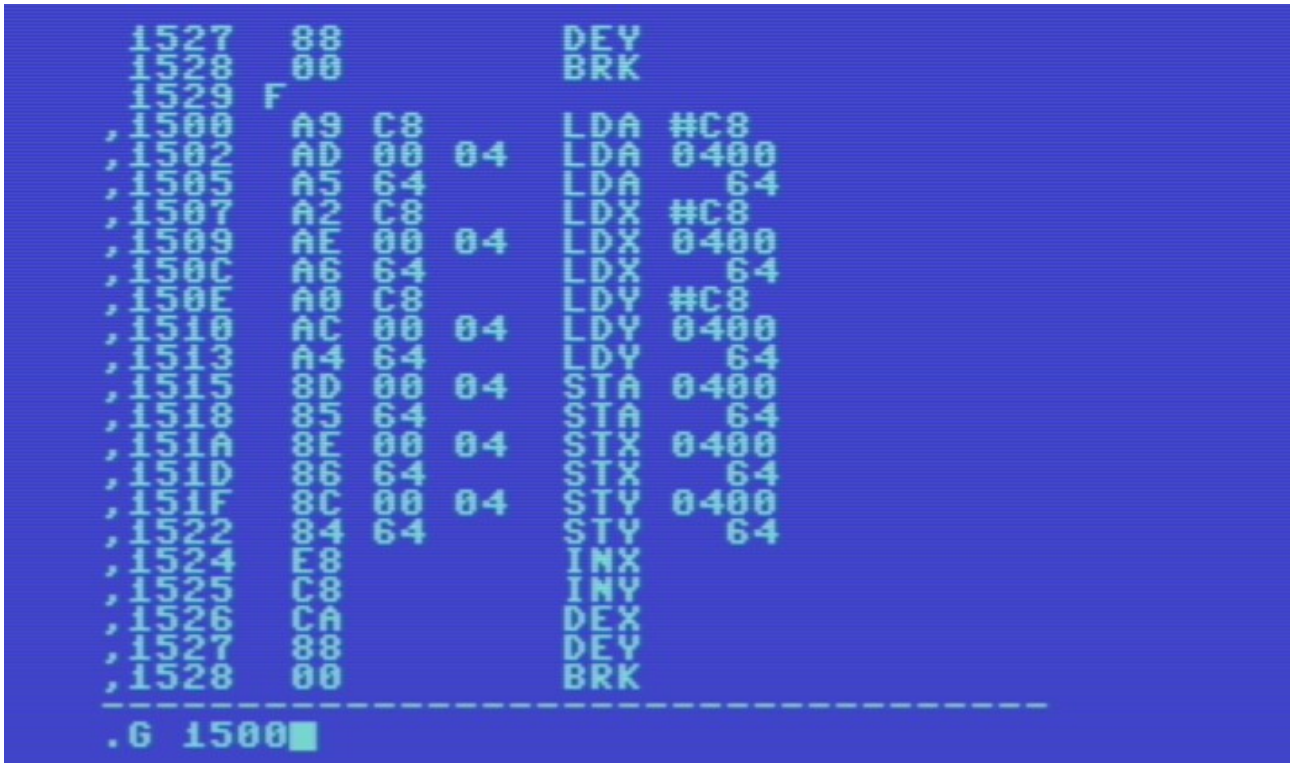
Es sollte Ihnen zeigen, wie die Assembler-Befehle sowie deren Parameter vom SMON in Maschinensprache übersetzt werden und wie sich unterschiedliche Typen von Parametern auf die generierten Befehlscodes auswirken.

Doch nun starten wir unser Programm doch einmal!

Dies erreichen wir durch den SMON-Befehl „G“ gefolgt von der Startadresse unseres Programms.

Geben wir also ein:

G 1500



Was zur folgende Anzeige führen sollte:

SMON meldet sich nach Beendigung unseres Programms wieder mit der Anzeige der aktuellen Registerinhalte (diese können bei Ihnen eventuell vereinzelt andere Werte enthalten)

Apropos Registerinhalte:

Der Inhalt des Akkumulators wird hier durch den Wert unter AC, jener des X Registers durch den Wert unter XR und jener des Y Registers durch den Wert unter YR angezeigt.

Wenn wir den Cursor nun beispielsweise auf die Cursorposition unter dem A von AC, auf die Position unter dem X von XR oder auf die Position unter dem Y von YR bewegen, können wir direkt Änderungen in dem jeweiligen Register vornehmen, die nach Betätigung der RETURN-Taste auch direkt übernommen werden.

Versuchen wir doch mal den Inhalt des Akkumulators auf den Wert $34 (dezimal 52), den Inhalt des X Registers auf den Wert $1A (dezimal 26) und den Inhalt des Y Registers auf den Wert $F8 (dezimal 248) zu ändern:

Mit dem Befehl „R“ können wir uns jederzeit die aktuellen Inhalte der Register anzeigen lassen, geben Sie also „R“ ein.

```
,1502   AD 00 04   LDA 0400
,1505   A5 64      LDA   64
,1507   A2 C8      LDX #C8
,1509   AE 00 04   LDX 0400
,150C   A6 64      LDX   64
,150E   A0 C8      LDY #C8
,1510   AC 00 04   LDY 0400
,1513   A4 64      LDY   64
,1515   8D 00 04   STA 0400
,1518   85 64      STA   64
,151A   8E 00 04   STX 0400
,151D   86 64      STX   64
,151F   8C 00 04   STY 0400
,1522   84 64      STY   64
,1524   E8         INX
,1525   C8         INY
,1526   CA         DEX
,1527   88         DEY
,1528   00         BRK
--------------------------------
.G 1500

  PC  SR AC XR YR SP  NV-BDIZC
;1529 B0 34 1A F8 F6  10110000
.R
```

Und wir sehen, dass die neuen Inhalte in die drei Register übernommen wurden:

```
,150C   A6 64      LDX   64
,150E   A0 C8      LDY #C8
,1510   AC 00 04   LDY 0400
,1513   A4 64      LDY   64
,1515   8D 00 04   STA 0400
,1518   85 64      STA   64
,151A   8E 00 04   STX 0400
,151D   86 64      STX   64
,151F   8C 00 04   STY 0400
,1522   84 64      STY   64
,1524   E8         INX
,1525   C8         INY
,1526   CA         DEX
,1527   88         DEY
,1528   00         BRK
--------------------------------
.G 1500

  PC  SR AC XR YR SP  NV-BDIZC
;1529 B0 34 1A F8 F6  10110000
.R

  PC  SR AC XR YR SP  NV-BDIZC
;1529 B0 34 1A F8 F6  10110000
.
```

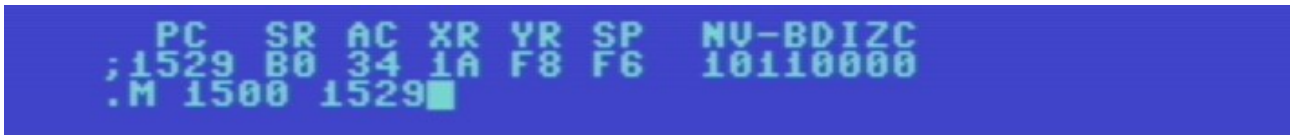
Ein weiterer hilfreicher Befehl ist „M“ Startadresse Endadresse

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

Unser Programm liegt wie gesagt im Arbeitsspeicher ab Adresse $1500 bis zur Adresse $1528, sehen wir uns daher einmal an, wie unser Programm im Arbeitsspeicher aussieht.

Der SMON verlangt bei der Angabe der Endadresse jedoch den um 1 erhöhten Wert, d.h. wir müssen hier statt $1528 den Wert $1529 angeben.

Geben wir also den Befehl M 1500 1529 ein:



Und erhalten folgendes Ergebnis:

Am Anfang erkennen wir hier den Befehlscode $A9 für den Befehl LDA #$C8 gefolgt vom Wert $C8, welcher ja die hexadezimale Darstellung des dezimalen Wertes 200 darstellt.

Das setzt sich nun Befehl für Befehl fort bis zum Ende des Programms.

Rechts neben der Adresse $1528 sieht man oben den Wert $00, welcher dem Befehlscode für den Befehl BRK entspricht und dem letzten Befehl in unserem Programm darstellt.

Direkt davor sieht man in der darüberliegenden Zeile ganz rechts den Wert $88, welcher den Befehlscode für den Befehl DEY repräsentiert.

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 Zeilen * 40 Spalten + Spalte 20) - 1 = 1483 oder hexadezimal $05CB.

Wir löschen mal den Bildschirm mit der Tastenkombination Shift + Pos1 und geben wie zuvor A 1500 ein:

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:

Löschen Sie wiederum den Bildschirm mit der Tastenkombination Shift + Pos1

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:

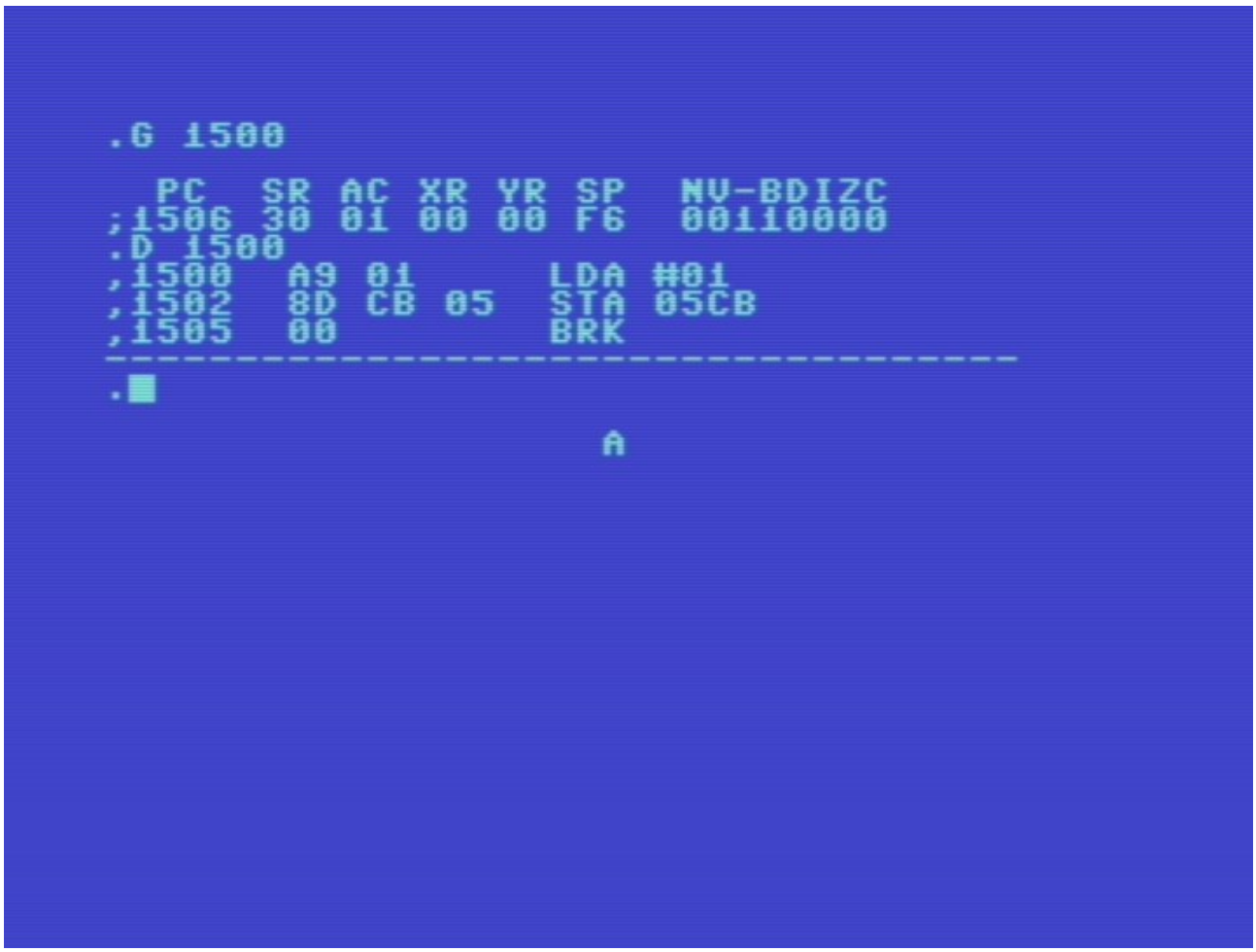
Wie erwartet wird in Zeile 12 / Spalte 20 der Buchstabe „A“ angezeigt:

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.

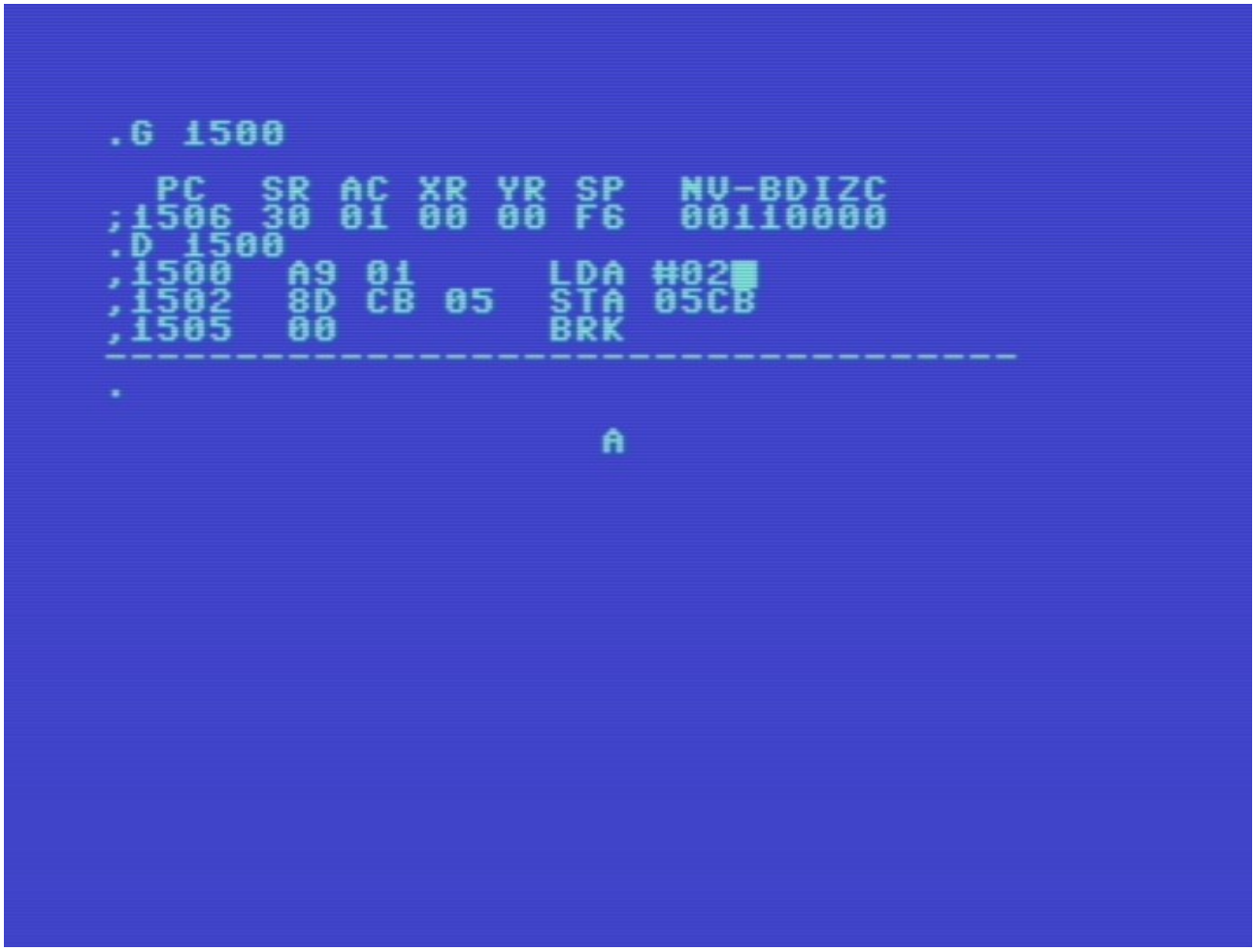
Es wird zunächst die erste Zeile unseres Programms angezeigt und durch Drücken der Leertaste kommt eine Zeile nach der anderen wieder zum Vorschein. Sobald die gestrichelte Linie nach dem BRK Befehl angezeigt wird, drücken Sie die Escape-Taste und Sie landen wieder im Befehlsmodus des SMON.



Durch den Befehl D 1500 haben wir den SMON erstmalig angewiesen, zu disassemblieren, d.h. die Zahlenwerte im Speicher wieder in Assembler-Befehle zurück zu verwandeln.

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

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



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

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.

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 diesesmal 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 diesesmal zur Abwechslung mal eine andere Adresse, nämlich die 12288 oder $3000.

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   NV-BDIZC
;C00B  B0 C2 00 00 F6   10110000
.D 3000
```

Wir weisen den SMON dadurch an, die Zahlencodes, die ab der Adresse $3000 im Speicher liegen, in Assembler-Befehle 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   NV-BDIZC
;300B  B0 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!

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:

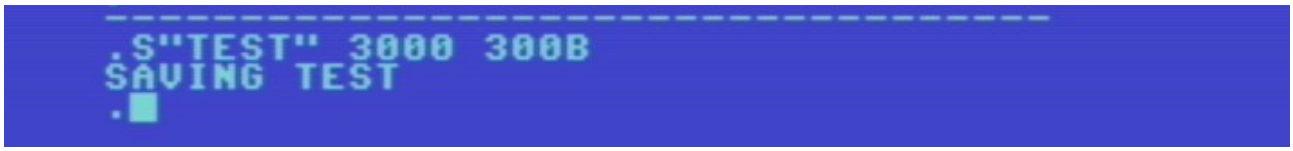
Ich habe hier einfach nur ein kurzes Programm mit ein paar Befehlen geschrieben, damit wir ein Programm haben, dass wir auf Diskette speichern und anschließend wieder von dort laden können.

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

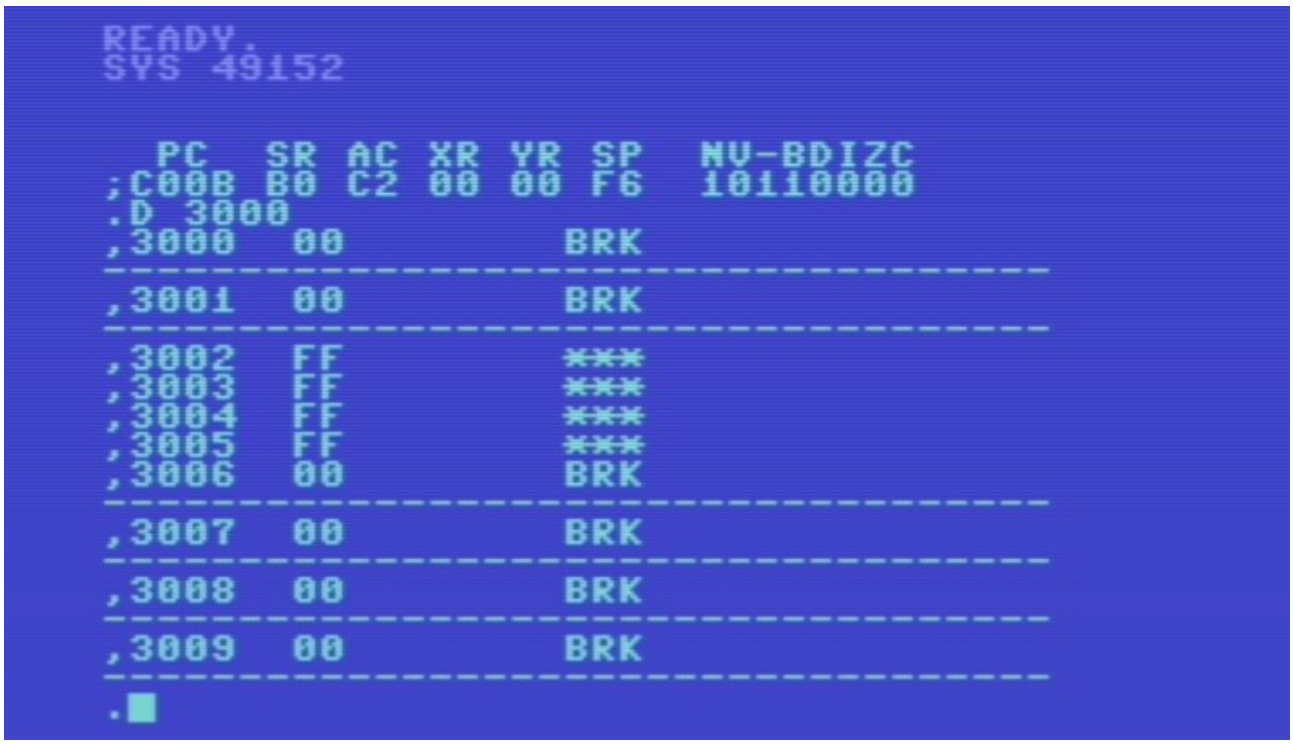
S„TEST“ 3000 300B



Geben Sie kein Leerzeichen zwischen „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.

Nun starten wir den C64 neu, laden den SMON erneut und lassen uns den Speicherbereich ab $3000 disassemblieren:



Ich habe hier obige Ausgabe erhalten, diese kann bei Ihnen unterschiedlich sein.
Das Disassemblieren soll an dieser Stelle den Zweck haben, den Inhalt des Arbeitsspeichers ab Adresse $3000 vor dem Laden des Programms anzuzeigen.

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.

Wir führen also den Befehl D 3000 aus und wie wir hier sehen, steht unser Programm tatsächlich wieder ab Adresse $3000 im Speicher (also nicht mehr der obige Inhalt der beim Disassemblieren vor dem Laden des Programms angezeigt wurde)

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.

Ein cooles Feature des SMON möchte ich Ihnen noch zeigen.

Der SMON bietet die Möglichkeit, ausgehend von einem Assembler-Programm, das sie geschrieben haben, DATA Zeilen (BASIC) zu generieren, welche das Assembler-Programm als Zahlenfolge (also als Maschinencode) enthalten.

Diese Zahlenfolge können Sie dann mittels FOR-Schleife auslesen und Byte und Byte in den Speicher schreiben. Ein solches Programm haben wir bereits am Ende des Abschnitts über die Maschinensprache kennengelernt. Solche Ladeprogramme werden im Zusammenhang mit Maschinensprache auch BASIC-Loader genannt.

Wenn Programme in Maschinensprache in Zeitschriften veröffentlicht werden, dann erfolgt das meist in Form eines solchen BASIC-Loaders und selten als Assembler-Code, da dies ja voraussetzen würde, dass der Anwender über einen Assembler verfügt.

Ein BASIC-Loader hingegen ist ein ganz normales BASIC-Programm, das man eingibt und mit RUN startet. Danach kann der BASIC-Loader mit NEW aus dem Speicher gelöscht werden, da er nur dazu diente, das in ihm enthaltene Maschinenprogramm in den Speicher zu bringen.

Nehmen wir das obige Programm als Ausgangsbasis.

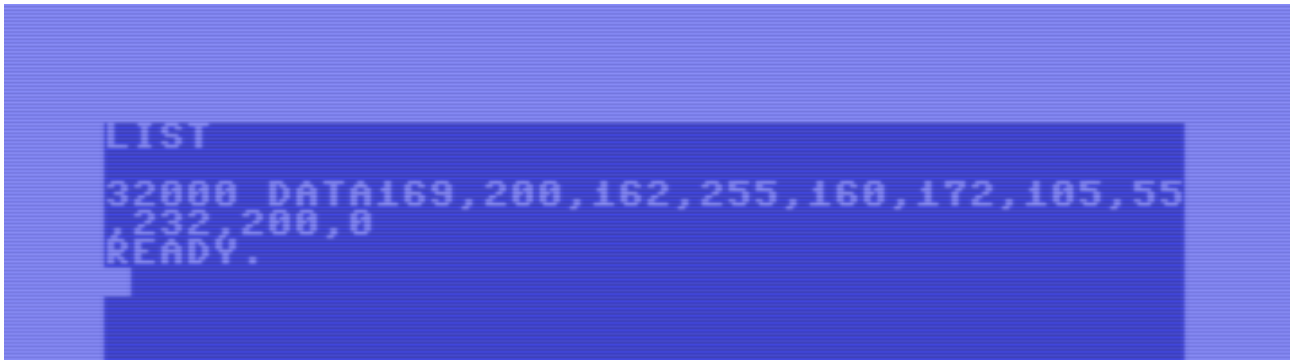
Der Befehl lautet B Startadresse Endadresse+1, in unserem Fall also

B 3000 300B

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

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:



Davor können wir nun die FOR-Schleife ergänzen, welche uns die Zahlenfolge Byte für Byte in den Speicher schreibt.

```
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:

```
LIST

10 FOR I=12288 TO 12298
20 READ A
30 POKE I,A
40 NEXT I
50 END
32000 DATA169,200,162,255,160,172,105,55
,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

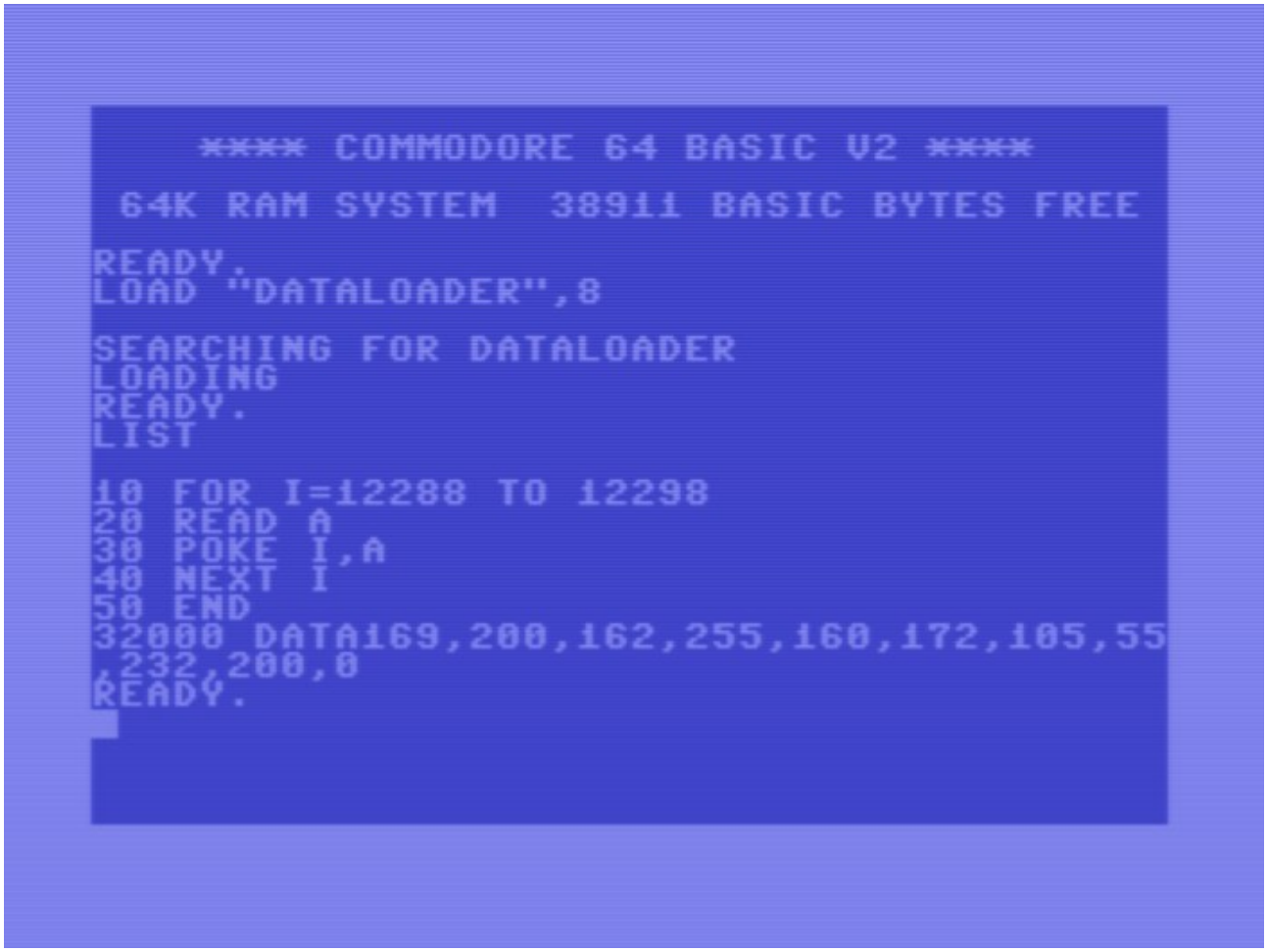
```
SAVE "DATALOADER",8

SAVING DATALOADER
READY.
```

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

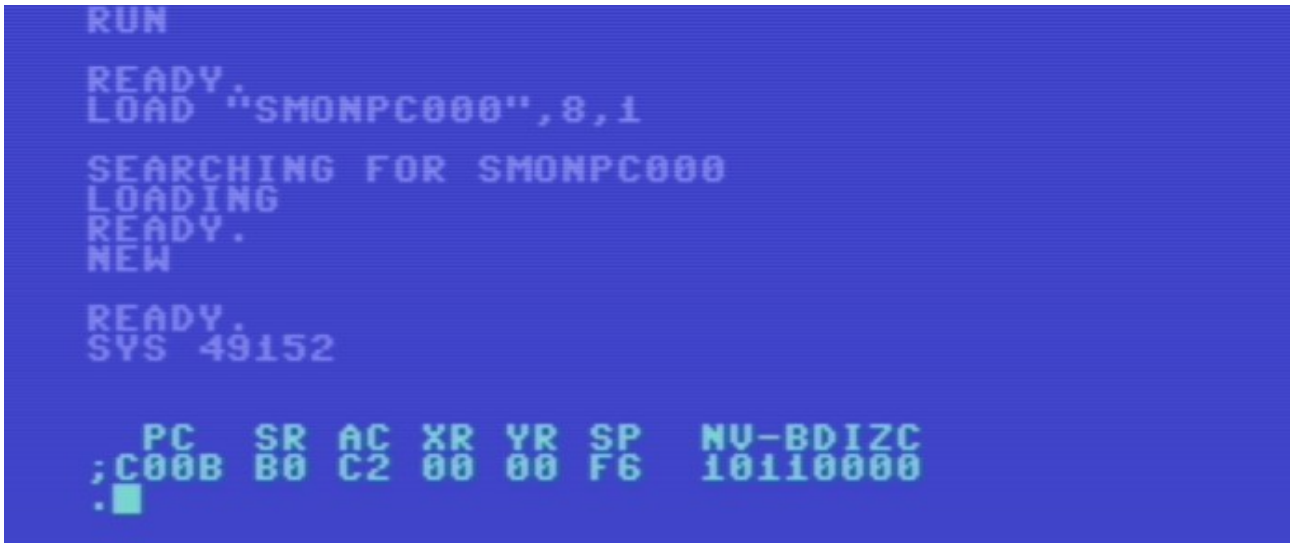
LOAD „DATALOADER“,8

wieder in den Speicher.



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

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



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

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.

Nun starten wir unser Programm mit SYS 12288 und sobald das Programm durchgelaufen ist, wollen wir nun 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.

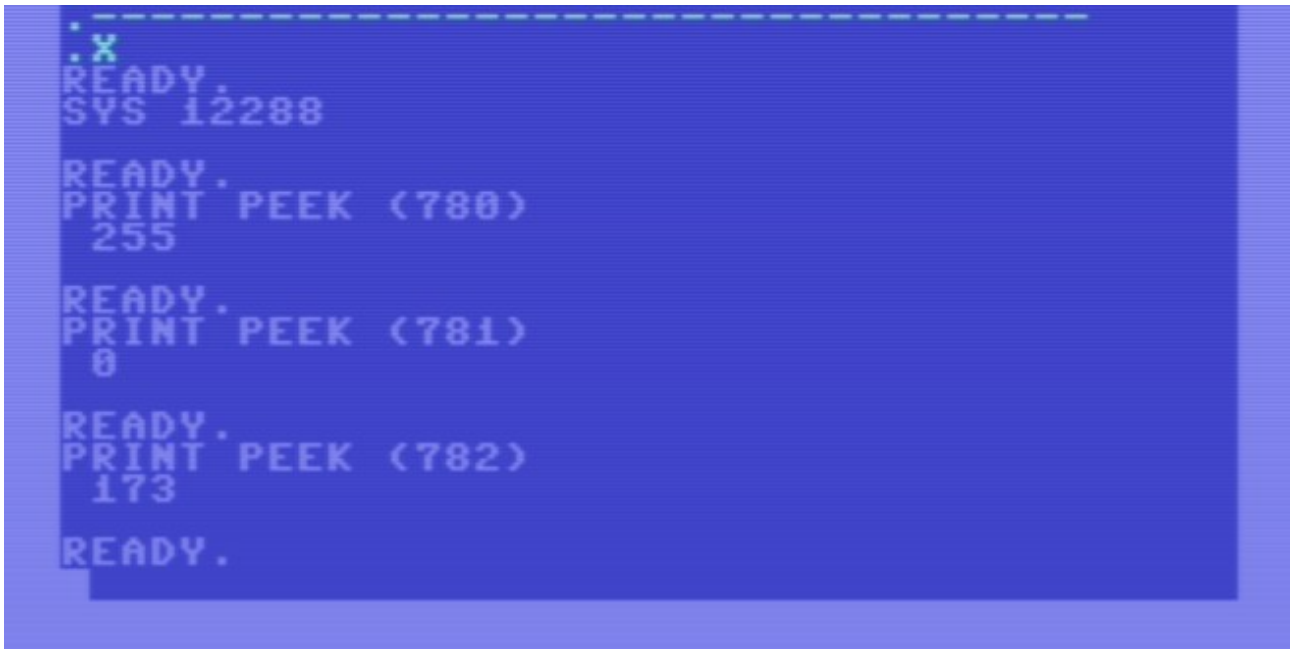
Weiter geht's mit dem nächsten Befehl, der den Inhalt des X Registers um 1 erhöht, d.h. wir erhalten 255 + 1 = 256.

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.



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.

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.

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.

Unmittelbare Adressierung (bereits verwendet)

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)

Absolute Adressierung (bereits verwendet)

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.

Zeropage Adressierung (bereits verwendet)

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.

Implizite Adressierung (bereits verwendet)

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

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.

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.

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.

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:

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

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 **B**ranch on **N**ot **E**qual (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:

```
.G 1500

  PC  SR AC XR YR SP  NV-BDIZC
;1509 33 C2 05 00 F6  00110011
.
```

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.

```
,1503  E0 05     CPX #05
,1505  D0 FB     BNE 1502
,1507  00        BRK
```

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.

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 Buchstaben 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 Buchstaben 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 Buchstaben 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.

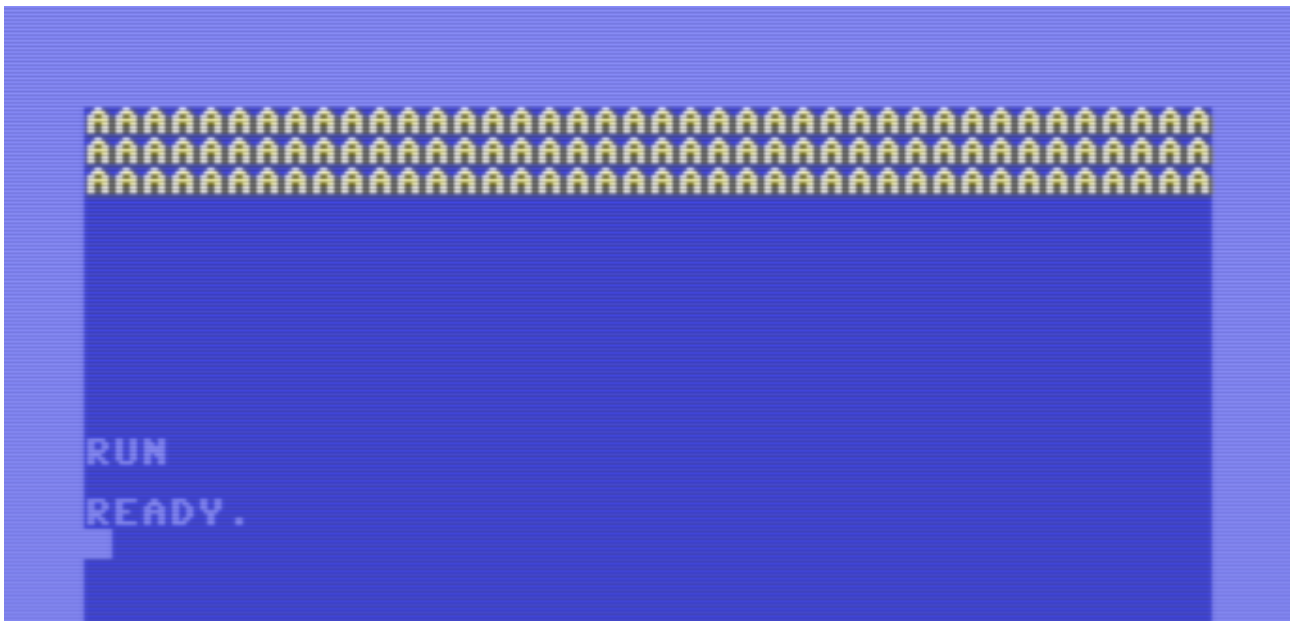
In der Speicherstelle 55297 steht der Farbcode des Zeichens rechts daneben und so weiter bis zur rechten unteren Bildschirmecke.

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 + Pos1, 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.

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 Farbcode 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 + Pos1, 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!

```
10 FOR I=0 TO 119
20 POKE 1024+I,1
30 POKE 55296+I,7
40 NEXT I
READY.
PRINT FRE(.)-65536*(FRE(.)<0)
 38854

READY.
PRINT 38911-38854
 57

READY.
```

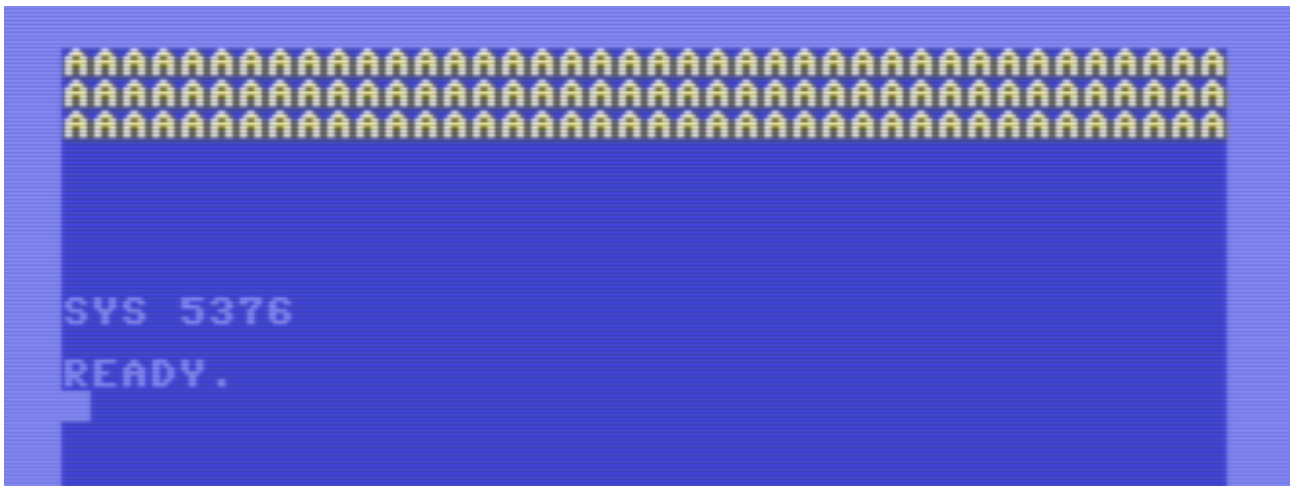
Starten wir das Maschinenprogramm zur Abwechslung mal wieder von BASIC aus. Dazu müssen Sie den BRK Befehl am Ende des Programms in den RTS Befehl ändern, damit wir nach dem Beenden des Programms im BASIC bleiben.

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

```
.G 1500

  PC  SR AC XR YR SP  NV-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 + Pos1, 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.

Dazu wechseln wir mit SYS 49152 wieder zu SMON zurück und erstellen mit

B 1500 1512

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

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.

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

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

Löschen wir wie gehabt wieder den Bildschirm mit Shift + Pos1, 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.

```
   PC   SR AC XR YR SP   NV-BDIZC
;C00B  B0 C2 00 00 F6   10110000
.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
--------------------------------
.S"THREELINES" 1500 1512
SAVING THREELINES
.
```

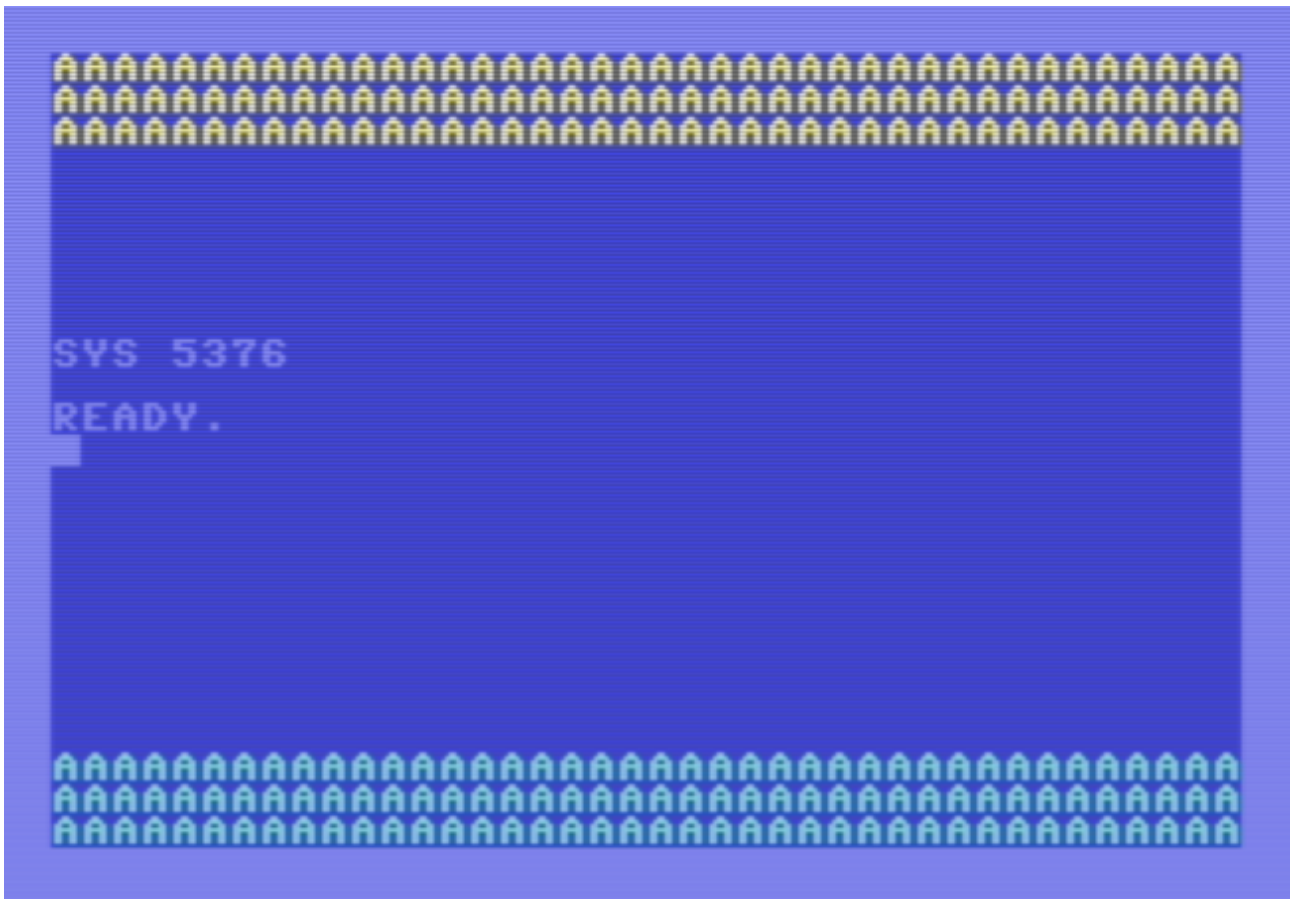
Starten Sie den C64 neu und laden das Maschinenprogramm wieder in den Speicher.

```
   PC   SR AC XR YR SP   NV-BDIZC
;C00B  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 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
--------------------------------
.
```

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 (Farbcode 3), angezeigt werden.

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:

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:

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 Befels 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.

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:

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 befüllt 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
,1530   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.

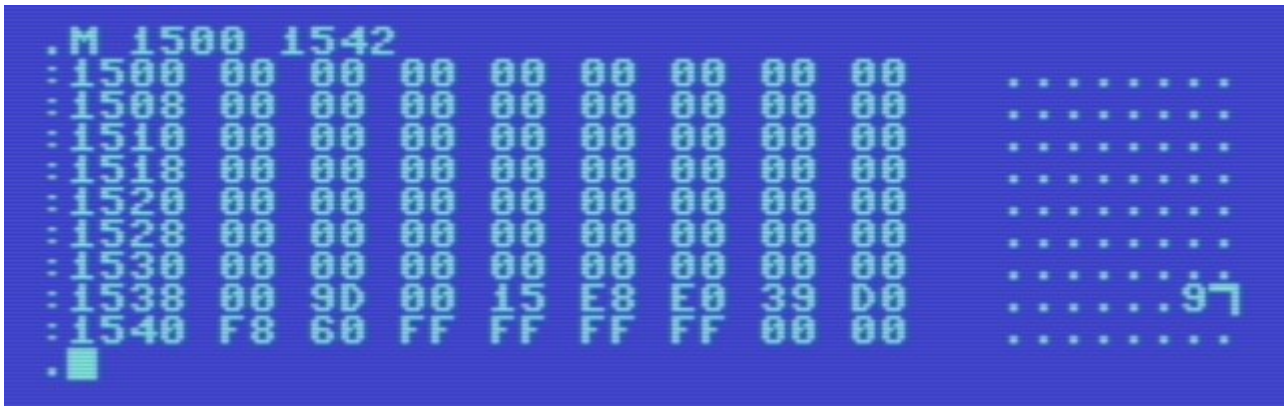
```
.D 1500
,1500  00        BRK
,1501  00        BRK
,1502  00        BRK
,1503  00        BRK
,1504  00        BRK
,1505  00        BRK
,1506  00        BRK
,1507  00        BRK
,1508  00        BRK
,1509  00        BRK
```

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



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 Löschschleife 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! :)

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.

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.

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.

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.

```
    **** 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  NV-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.

Geben Sie nun ab Adresse $1500 folgende Befehle ein:

```
   PC  SR AC XR YR SP  NV-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)
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  NV-BDIZC
;150D 30 28 00 00 F2  00110000
.
```

Durch den folgenden Befehl können wir uns den Inhalt des Stacks anzeigen lassen:

```
.M 0100 01FF
```

```
:01C8 00 00 FF 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  .1......
:01E8 B8 E6 A7 E6 24 00 20 0A  ....$. .
:01F0 C4 11 C3 28 1E 14 0A 46  -.-(...F
: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)	unebekannt

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)

Im Zusammenhang mit dem Stack möchte ich Ihnen abschließend noch folgende zwei Befehle vorstellen:

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.

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.

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:

```
.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 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
--------------------------------
,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.

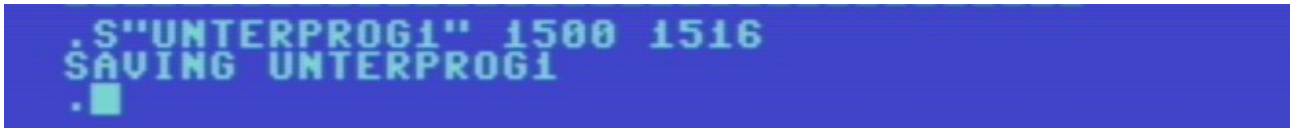
In diesem Fall ist das der Befehl RTS, wodurch unser Programm beendet wird.

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 Farbcode 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.

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

```
.S"UNTERPROG2" 1500 1519
SAVING UNTERPROG2
.
```

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.

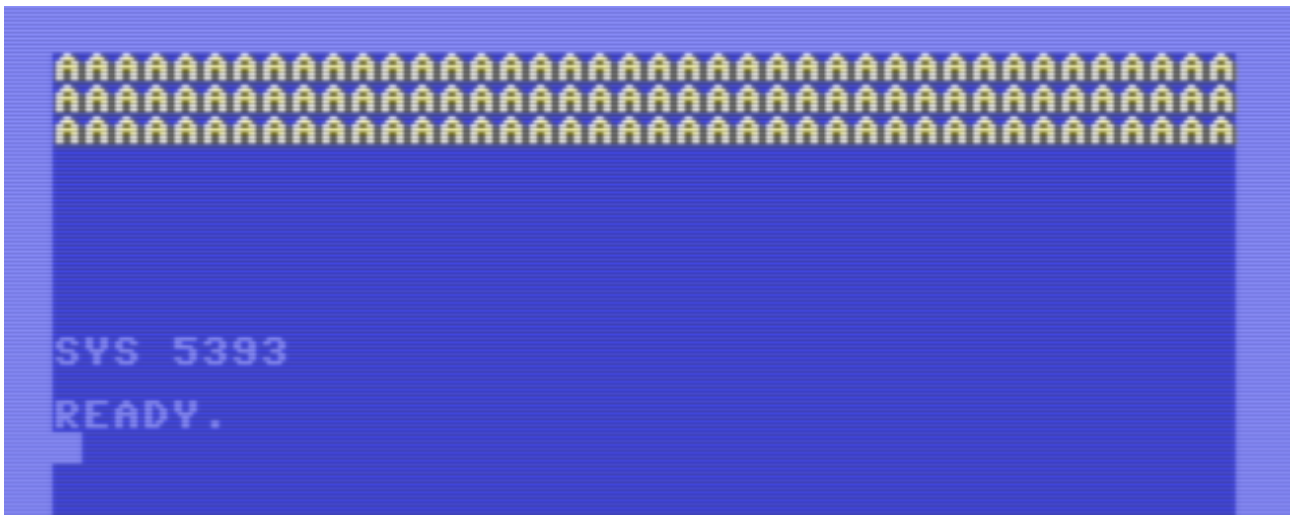
Nun wird durch den Befehl TYA der Inhalt des Y Registers, welches ja den Farbcode enthält, in den Akkumulator übertragen.

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.

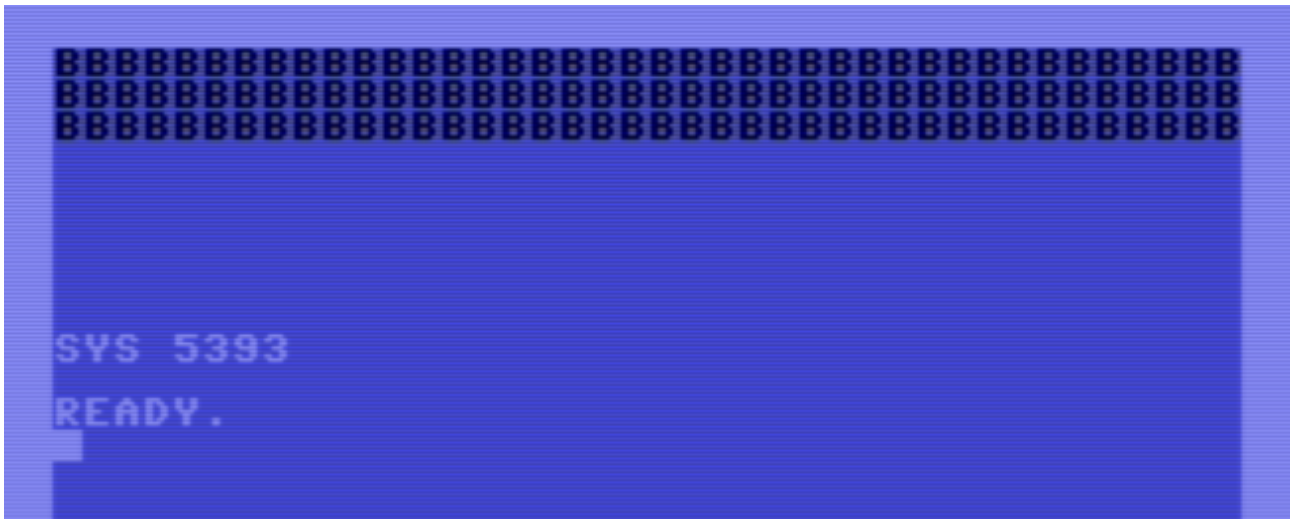
Der Rest läuft ab wie vorhin, der Inhalt des X Registers wird um 1 erhöht, mit dem Wert $78 (dezimal 120) verglichen und solange dieser Wert nicht erreicht ist, wird zu der Programmadresse $1502 gesprungen.

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.

```
.A 1511
 1511   A9 03      LDA #03
 1513   A0 03      LDY #03
 1515   20 00 15   JSR 1500
 1518   60         RTS
```

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC



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
```

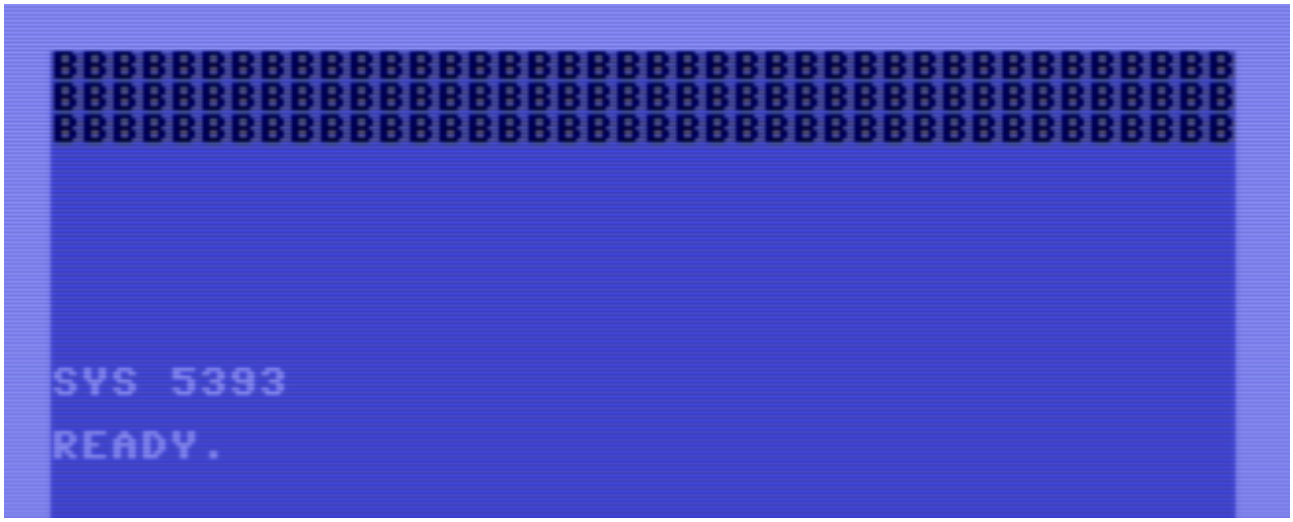
Speichern Sie das Programm unter dem Namen „UNTERPROG3“ mit folgendem Befehl ab.

```
.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.

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC



SYS 5393

READY.
```

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:

```
,1518  20 E4 FF   JSR FFE4
,151B  F0 FB      BEQ 1518
```

bzw.

```
,1524  20 E4 FF   JSR FFE4
,1527  F0 FB      BEQ 1524
```

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!

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 Farbcode 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“

```
 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“.

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.

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“

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.

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.

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 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.

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 Cursortasten 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.

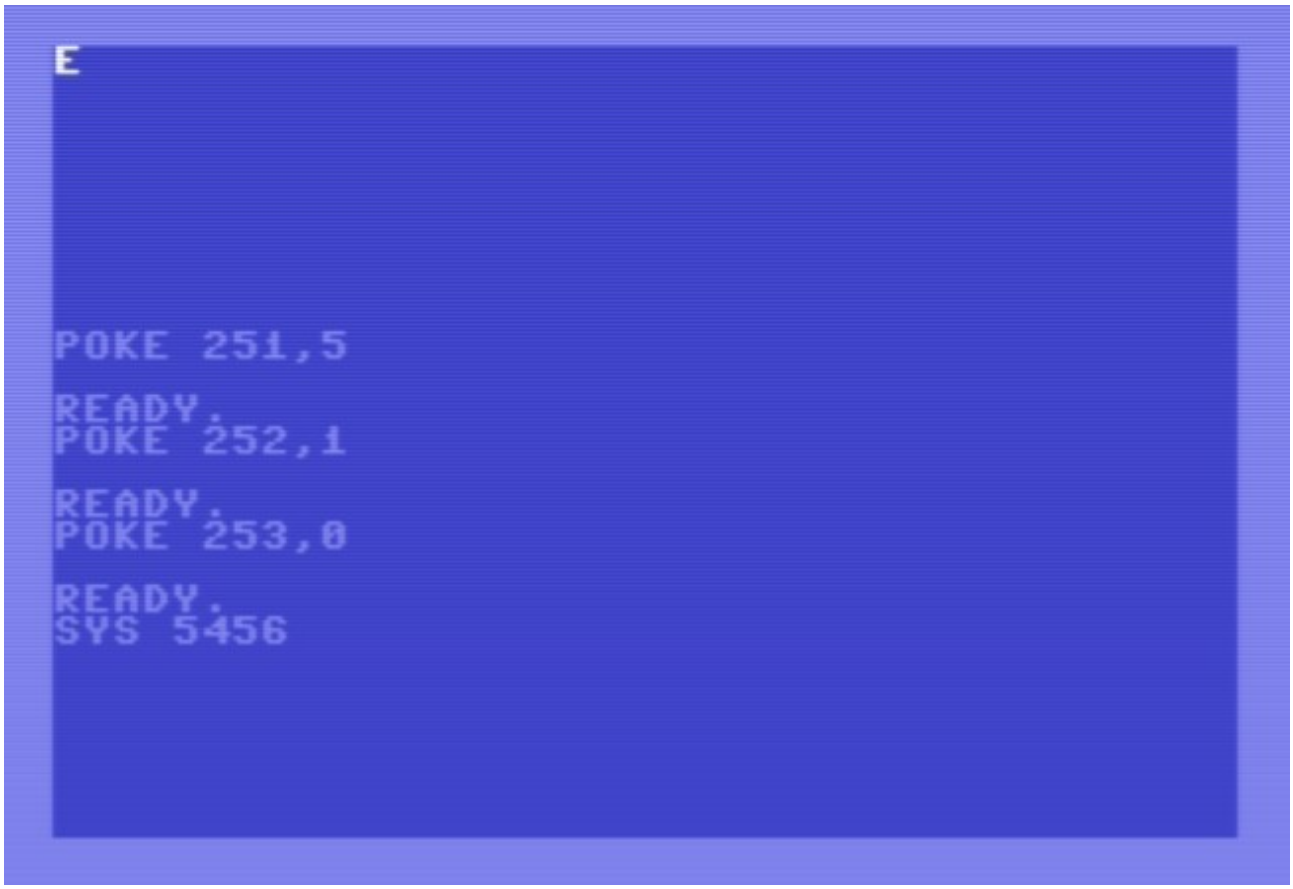
Das ist richtig, aber wir haben die Möglichkeit, das Programm nicht an der Adresse $1544 zu starten, sondern erst an der Adresse $1550 (dezimal 5456).

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.

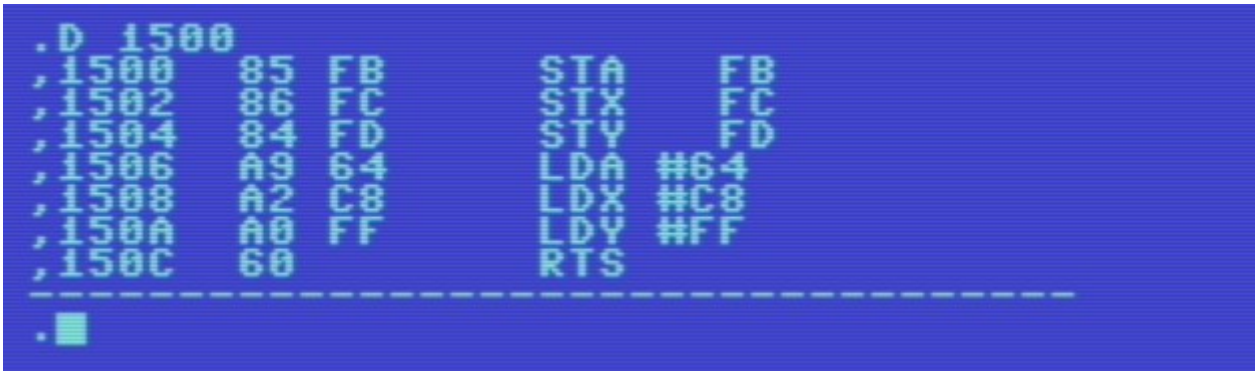
Das heißt, man könnte diese Vorgehensweise auch zur Parameterübergabe an ein Maschinenprogramm nutzen und es wäre eine alternative Herangehensweise an die Parameterübergabe in den CPU-Registern, welche bereits beschrieben wurde.

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.

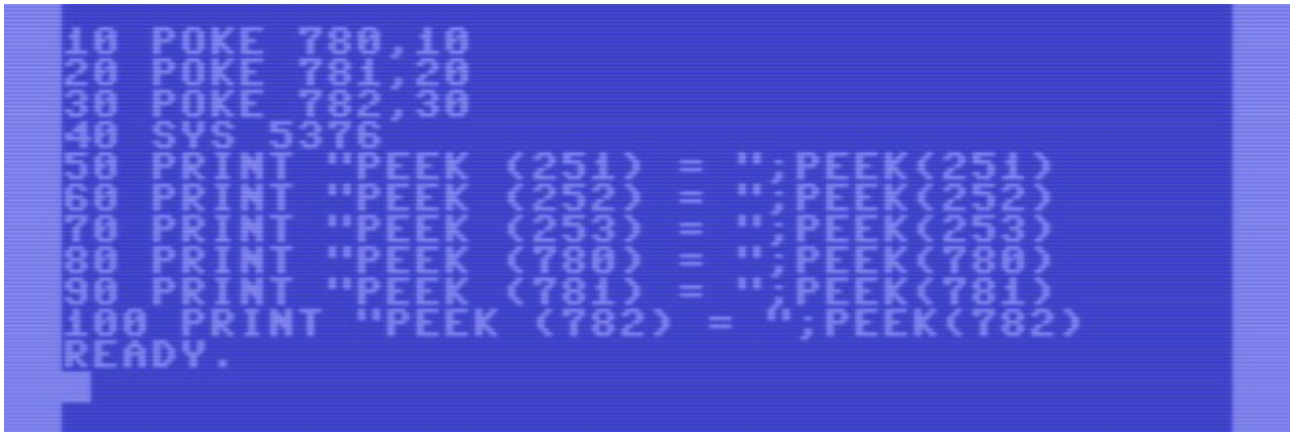
Geben Sie folgendes Programm ab Adresse $1500 ein:



und speichern es mit dem Befehl S“REGTESTASM“ 1500 150D

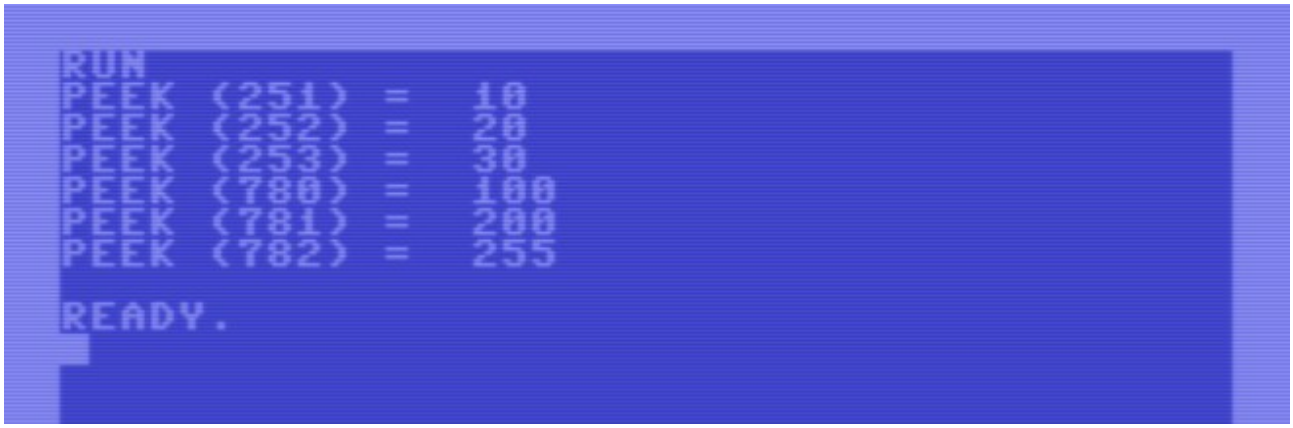
auf Diskette.

Dann wechseln Sie zurück zu BASIC und geben folgendes Programm ein:



Speichern Sie es unter dem Namen „REGTEST“ ab.

Nach dem Start mit RUN sollten Sie folgendes Ergebnis erhalten:



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.