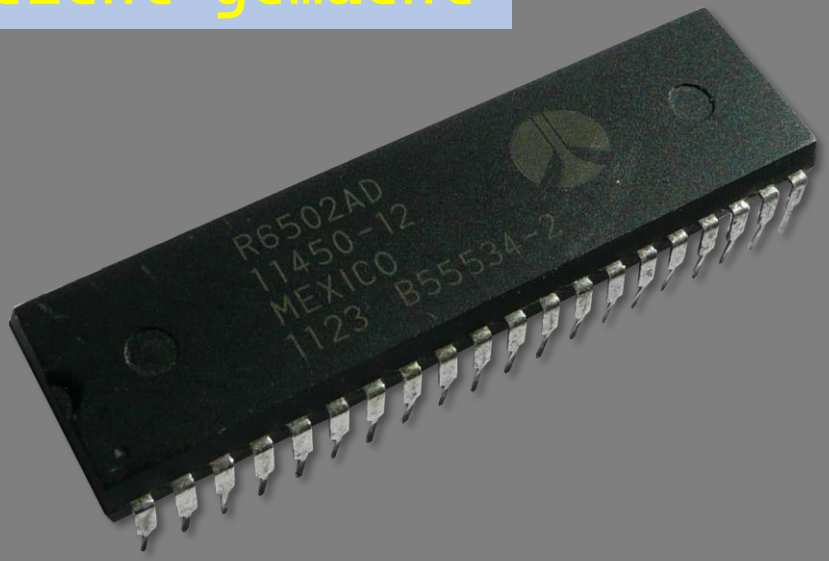
Der C64

BASIC
und
ASSEMBLER

leicht gemacht



Alle Beispiele mit dem
SMON-Assembler

Teil 1

Keine Angst vor Bits & Bytes

Erik Bartmann

Dieses Dokument	
Titel	Der C64 - BASIC und Assembler
Thema	Eine kleine Einführung in den Commodore C64
Erstellt am	13.08.2024
Erstellt von	Erik Bartmann
Version	Teil 1 - Version 1.03

Dein Ansprechpartner	
Name	Erik Bartmann
E-Mail	erik.bartmann@yahoo.de
Internet	https://erik-bartmann.de/

Copyright 2025 – Erik Bartmann

Inhaltsverzeichnis

Vorwort

Auf die Gefahr hin, dass ich mich bei der jüngeren Generation lächerlich mache, was mich aber recht wenig bis überhaupt nicht stören würde, bin ich zu dem Schluss gekommen, etwas über die Programmierung in Maschinensprache zum Besten zu geben. Vor ca. *45* Jahren habe ich mit dem *Commodore CBM 3032* angefangen und mir darauf einen *Apple IIe* zugelegt. Ein *Wahnsinns-Teil*, bei dem die Programmierung wirklich Spaß bereitet hat. Dort habe ich die ersten Berührungspunkte mit der Maschinensprache für den *6502*-Prozessor gemacht, was aber im Laufe der Jahre, als dann der PC angesagt war, in Vergessenheit geriet. Zwar habe ich noch einmal einen Lehrgang für den *Z80* -Prozessor besucht, doch dann war die Zeit endgültig vorbei.

Durch das Aufräumen im Keller bin ich dann irgendwann auf meine alten Schätze gestoßen und habe mir - da ich nie einen besessen habe - einen Commodore C64 zugelegt. Ein Arbeitskollege wollte ihn samt dem Diskettenlaufwerk in *„Gute Hände"* geben und deshalb war er bei mir an der richtigen Adresse. Was lag also näher, mein Vorhaben, mich noch einmal mit der Maschinensprache des *6502* zu befassen, in die Tat umzusetzen. Der *Commodore C64* hat einen Prozessor vom Typ *6510*, der den gleichen Befehlssatz besitzt, wie der *6502* und lediglich über einen zusätzlichen *6-Bit* Datenkanal verfügt.

Abbildung 1 - Der Commodore C64 - auch Brotkasten genannt

Vielleicht kann ich den einen oder anderen für die Maschinensprache begeistern, obwohl dieser veraltete Typ sicherlich kaum noch Verwendung findet. Es macht halt einfach tierisch Spaß, sich mit der Materie auseinander zu setzten und das Gefühl zu haben, seine Maschine zu verstehen und mehr oder weniger zu beherrschen. Wer kann schon von seinem Windows-PC behaupten, er wüsste genau, was das vor sich geht. Es spielen sich so viele Dinge im Hintergrund ab und es werden ständig irgendwelche Patches und Updates eingespielt, dass wir mit Sicherheit sagen können, dass der PC nicht wirklich unter unserer Kontrolle steht. Wenn ich an das IT-Unternehmen denke, in dem ich gerade arbeite, dann muss ich manchmal mit einem lachenden

und weinenden Auge an die Spezialisten denken, die sich mit hochkomplexer Software herumschlagen (müssen). Natürlich werden die Fachkräfte hinsichtlich der im Einsatz befindlichen Software geschult, doch es kommen immer einmal Situationen vor, wo selbst der fähigste Kollege quasi *auf dem Schlauch steht*. Es geht weder vor noch zurück und im Endeffekt ist der Hersteller der betreffenden Software oder auch Hardware gefragt, um helfend einzugreifen. Das ist aber manchmal aufgrund unzähliger Hard- bzw. Software-Komponenten in unterschiedlichen Konfigurationen nicht immer so einfach. Der Hersteller fordert diese oder jene Log-Files an, um sich ein Bild vom vorherrschenden Zustand des Kundensystems mit den gerade vorhandenen Problemen zu machen. Das gleicht einem Ping-Pong Szenario, wobei die Nerven der betreffenden Personen - meist die der Kunden - schon mal blank liegen. Da werden System-Resets durchgeführt, Patches eingespielt und hier und da ggf. an diversen Konfigurationen *herumgespielt* und jeder hofft, dass innerhalb einer verträglichen Zeit das System wieder auf die Beine kommt. Wenn es dann wieder funktioniert ist die Freude groß und nicht selten schauen sich die beteiligten Personen anschließend fragend in die Augen und sinnen darüber nach, woran es denn nun wirklich gelegen hat. Eine befriedigende Antwort liegt dann meistens in weiter Ferne und das abschließende Statement lautet dann: *„Na Hauptsache, dass es wieder funktioniert!*" Bis zum nächsten Crash, bei dem das Spiel von vorne beginnt. Die Beherrschbarkeit der Systeme ist teilweise zu einem Wunschdenken geworden.

Ist es nicht faszinierend zu sehen, dass die damaligen Entwickler der Betriebssysteme, die sich fest im ROM, also im Festspeicher befanden, diese programmiert haben, ohne, dass später unzählige Fehler gefunden wurden und Patches notwendig waren, was zur damaligen Zeit sowieso nicht praktikabel war. Heutzutage schaut das anders aus. Natürlich ist die Komplexität der Systeme enorm gewachsen und mit Maschinen wie dem *C64* oder *Apple II* nicht zu vergleichen, doch es ist trotzdem für mich fast unglaublich, dass es damals so gut funktioniert hat. Das kann vielleicht nur jemand so ausdrücken, der die damalige Zeit hautnah miterlebt hat, doch sei es drum.

Um ein bisschen von dem ehemaligen Gefühl der Faszination zu vermitteln, einen Prozessor zu programmieren, habe ich mich zu diesem Schritt entschieden. In erster Linie schreibe ich das Ganze für mich und hoffe dennoch, dass der eine oder andere sich mitreißen lässt. Ich möchte hier nicht das ultimative Werk für den C64 schaffen, was sowieso illusorisch wäre, denn es gibt *soooo* viel Literatur, die frei zum Download angeboten wird und allemal tiefgründiger sind. Es soll einfach nur ein lockerer Einstieg in die Thematik *Commodore C64* sein. Mal schauen, was dabei rumkommt.

Einleitung

Um sich mit der Programmierung des *Commodore C64*, sei es in der Programmiersprache *BASIC*, in *Maschinensprache* oder in welcher verfügbaren Sprache auch immer auseinanderzusetzen, ist es noch nicht einmal nötig, einen eigenen *C64* zu besitzen, obwohl sicherlich viele im Internet zum Verkauf angeboten werden. Es stehen einige sogenannte *Emulatoren* zur Verfügung, die einen *C64* in seinen unterschiedlichen Funktionen nachbilden und emulieren. Der bekannteste Vertreter ist sicherlich der *VICE-Emulator*, der auch andere Computer wie z.B. den *VC20* und einige andere emuliert.

Abbildung 2 - Der C64 im VICE-Emulator

Das schaut exakt so aus, als wenn ein wirklicher und realer *C64* seine Arbeit verrichten würde. Mit dem *VICE-Emulator* können wir genauso gut arbeiten, wie mit dem *Brotkasten*.

	Wie kann VICE heruntergeladen werden?
https://vice-emu.sourceforge.io/	

Soweit die Grundvoraussetzungen, um mit der Programmierung in Maschinensprache beginnen zu können. Eine Kleinigkeit fehlt jedoch noch. Computer wie der *Apple II* oder auch die Rechner der *Commodore CBM-Serie (PET/CBM)* haben ein eingebautes *Monitor-Programm*, das sich *MLM* (***M**achine **L**anguage **M**onitor*) nennt. Darüber ist es mehr oder weniger möglich, direkt in *Maschinensprache* bzw. *Assembler* zu programmieren. Auf den Unterschied werde ich zu gegebener Zeit noch genauer eingehen. Ach ja, beinahe hätte

ich es vergessen. Der *C64* verfügt eben *nicht* über ein derartiges *Monitor-Programm*, so dass wir auf externe Komponenten in Form von weiterer *Software* oder *Erweiterungsmodule*, auch *Cartridges* genannt, zurückgreifen müssen. Wenn Du den *VICE-Emulator* verwendest, hast Du einen entscheidenden Vorteil. Du musst Dir nicht extra ein Erweiterungsmodul kaufen, sondern kannst eine Software-Komponente nutzen, die das entsprechende Modul ebenfalls *emuliert*. Ich verwende in diesem Buch das Assemblerprogramm *SMON*, das zum freien Download angeboten wird. Die Links der Software, die ich hier verwende, findest Du im Anhang oder auch auf meiner Internetseite. Neben der Maschinensprache werde ich auch hier und da ein wenig auf die Programmiersprache *BASIC* eingehen. Das sollte einfach dazugehören.

Wenn VICE genutzt wird, dann ist es sinnvoll, sich vorher mit ein paar Tastenbelegungen auseinanderzusetzen. Ich zeige nachfolgend das Layout, das man sich gut einprägen sollte!

Abbildung 3 - VICE-Layout (DE) - Quelle: vice-emu.sourceforge.io

Unter der folgenden Internetadresse sind weitere Hinweise zu finden.

	The Versatile Commodore Emulator
https://vice-emu.sourceforge.io/	

Mein didaktischer Ansatz

Ich werde in diesem kleinen Büchlein etwas von der „normalen" Struktur und Vorgehensweisen anderer Bücher abweichen und nicht nach *Schema-F* vorgehen. Wenn ich etwas überhaupt nicht mag, sind es Fakten, die stur heruntergebetet werden. Natürlich geht es nicht ohne ein gewisses Maß an Auflistung von Details, die zum Verständnis einer neuen Materie dienen. Ich möchte so vorgehen, dass ich - und das ist nur meine Meinung und muss nicht allgemeingültig für jeden passend sein - genau zum richtigen

Zeitpunkt das liefern und präsentieren, wenn es erforderlich ist. Das mag vielleicht hier und da etwas unstrukturiert erscheinen, doch wer es nicht mag, der möge einfach aufhören zu lesen. Ganz einfach eben!

Echte Hardware

Für mich ist es beim Schreiben natürlich einfacher, wenn ich eine Emulator-Software verwende, um die zahlreichen Screenshots zu erstellen und Tests zu machen. Doch ein richtiges Gefühl von Nähe zu einem C64 kommt natürlich nur auf, wenn ein derartiger Computer vor einem steht und man ihn wahrlich anfassen kann. Ich rate also jedem, der wahrlich in Kontakt mit diesem Gerät kommen will, sich einen C64 zu ersteigern. Dazu bedarf es dann nicht unbedingt eines alten Röhrenmonitors, denn es gibt zahlreiche Konverter, um das C64-Signal in ein HDMI-Signal zu wandeln und dann den TFT auf dem Schreibtisch zu nutzen. Der C64 besticht in seinem Erscheinungsbild durch seine Schlichtheit und Schönheit. Da sich sein Betriebssystem in einem ROM befindet, kann er, falls er nicht defekt ist - sehr schnell hochfahren, was nicht mehr wie *2* Sekunden in Anspruch nimmt. Es gibt also nicht die leidigen Themen wie zum Beispiel Konfigurationsdateien, die erst abgearbeitet werden müssen, um einen Computer beim Starten den letzten Schliff zu geben, um dann möglicherweise in ein Problem zu laufen, was den Start in letzter Sekunde noch vereitelt.

Eine wunderschöne Eigenschaft des C64 hinsichtlich der Eingabe von Befehlen ist der sogenannte *Fullscreen-Editor*. Sehe viele Computer der damaligen Zeit besaßen als Kommandozeilenschnittstelle lediglich eine Art von Terminal, in dem man etwas in einer einzigen Zeile eintippen konnte. Eine erneute Eingabe war dann nur in einer nachfolgenden Zeile möglich, denn es war nicht vorgesehen, zum Beispiel mit den Cursor-Tasten über den Bildschirm zu navigieren und an schon gemachten Eingaben Änderungen vorzunehmen. Das alles kann aber der C64 mit seinem *Fullscreen-Editor*. Es ist möglich, sich mit den Cursor-Tasten frei zu bewegen und nach gemachten Anpassungen erneut die Eingabe mit der *RETURN*-Taste zu bestätigen.

Grundlagen

Wenn wir heutzutage Computer programmieren, dann erfolgt das meistens in einer sogenannten *Hochsprache*. Darunter fallen Programmiersprachen wie *C/C++*, *C#*, *VB.NET*, um nur sehr wenige zu nennen. Um sich dem Computer in irgendeiner Form mitzuteilen, müssen dem Programmierer Werkzeuge an die Hand gegeben werden, die es ihm ermöglicht, ein bestehendes Problem in irgendeiner Weise zu beschreiben und entsprechende Anweisungen für den Computer zu formulieren. In den Anfängen der Computertechnik erfolgte dies über die *Maschinensprache*, die quasi eine *Low-*

Level-Programmierung darstellt und lediglich aus bestimmten - für den Menschen schwer lesbaren - Codes bestand, die der Prozessor der jeweiligen Prozessorfamilie *verstand*. Eine *Hochsprache* ist unabhängig vom darunterliegenden Prozessor und kann auf unterschiedlichen Maschinen zum Einsatz kommen. Eine Programmiersprache wie z.B. *BASIC*, die die Standardprogrammiersprache der frühen Computer war, setzte sich aus vielen lesbaren und verständlichen Kommandos zusammen, so dass die Erstellung eines Programms für einen Computer wie den *C64* an eine Aneinanderreihung von Befehlen glich, die an eine Militärsprache erinnert. Kurz und knapp und dennoch unmissverständlich. Die Befehlssprache kommt aus dem amerikanischen und wurde aus Gründen der Kompatibilität nicht in die jeweiligen Landessprachen übersetzt. Da gibt es Befehle wie z.B. *PRINT* oder *INPUT*, die für *Ausgabe* oder *Eingabe* stehen. Das ist für uns Menschen allemal verständlicher wie irgendein ominöser Code. Eine Anweisung einer Hochsprache kann sich aus mehreren Maschinenbefehlen zusammensetzen, wogegen eine Anweisung in Maschinensprache oder Assembler wirklich nur aus einem einzigen Maschinenbefehl besteht.

Aber hey, warum wurde dann überhaupt zu Zeiten von *BASIC* in *Maschinensprache* programmiert und nicht in verständlichen Befehlen, wie sie BASIC zur Verfügung stellte? Die Antwort lautet: Maschinensprache ist *rattenschnell*. Es ist *die* Sprache, die der Prozessor direkt versteht und nicht erst durch einen *Dolmetscher* übersetzt werden muss. Man kann davon ausgehen, dass ein Maschinenprogramm ca. *10*- bis *1000*-mal schneller in der Ausführung ist, als ein Programm in BASIC für die gleiche Aufgabe.

BASIC benutzt eine Syntax, die für uns Menschen recht verständlich ist, für den Prozessor jedoch *böhmische Dörfer* darstellen. Wir haben die gleichen Probleme mit der Maschinensprache, wie der Prozessor mit BASIC. Sie sind nicht verständlich, weil sie nicht dem täglichen Sprachvorrat entspringen. Wir haben den Vorteil, dass wir den Code der Maschinensprache erlernen können. Der Prozessor hat die Möglichkeit nicht und benötigt deswegen einen *Dolmetscher*, der in der Funktion eines Vermittlers arbeitet. Es liegt in der Natur der Sache, dass dieser Vorgang Zeit in Anspruch nimmt und deswegen geht die Abarbeitung eines BASIC-Programms immer langsamer vonstatten, als bei Maschinensprache. Auch heutzutage werden zeitkritische Anwendungen noch direkt in Maschinensprache programmiert, wobei aber die Programmierung unter *Windows* aufgrund bestimmter Sicherheitsaspekten nicht so ohne weiteres möglich ist.

Wie arbeitet ein Prozessor

- Der 6502-/6510 und das Bussystem
- Die Register des Prozessors
- Befehle und Daten

Wie arbeitet ein Prozessor?

Innerhalb eines Prozessors herrscht absoluter *Dualismus*. Da gibt es kein *Vielleicht* oder *Möglicherweise*. Ganz so, wie wir Menschen uns des Öfteren verhalten und keine Schattierungen oder Abstufungen erkennen, gibt es nur ein *Ja* oder *Nein*. Diese beiden Zustände müssen natürlich irgendwie in elektrische Signale überführt werden, so dass es nur ein *Strom fließt* oder ein *Strom fließt nicht* gibt.

Diese beiden Zustände repräsentieren eine bestimmte Art und Weise der Informationsinterprätation. Würde man das auf einen einfachen Schaltkreis mit einem Schalter, einer Batterie und einer Lampe übertragen, bedeutet dies, dass die Lampe *AUS* (Strom fließt nicht) oder *AN* (Strom fließt) ist. Diese beiden Zustände benötigen zur Darstellung lediglich zwei unterschiedliche Symbole, *0* und *1*. Auf der nachfolgenden Abbildung ist das mit dem gerade erwähnten Schaltkreis sehr gut zu erkennen, wobei *L* die Lampe und *S* der Schalter ist.

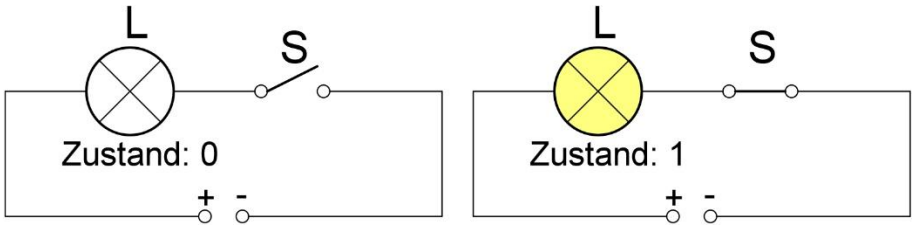


Abbildung 4 – Eine einfache Logikschaltung

Diese beiden Zustände werden in der Literatur zeitweise mit unterschiedlichen Bezeichnungen versehen.

- Ein - Ja - Wahr - 1
- Aus - Nein - Falsch - 0

Wenn es um die Benennung derartiger Werte geht, dann hat dieser *Dualismus* eine besondere Kennzeichnung erhalten. Du wirst gleich Werte kennenlernen, die sich nur aus *1* bzw. *0* zusammensetzen. Diese Werte werden *Binär-Zahlen* genannt. *Binär* bedeutet *Zwei* und bezieht sich auf die dazugehörige Basis, die den Werten zugrunde liegt. Wir arbeiten täglich im sogenannten *10er-System*, was für uns kein Problem darstellt und wir auch bei irgendwelchen Berechnungen nicht darüber nachdenken. Es ist uns in Fleisch und Blut übergegangen. Hierzu ein kleines Beispiel. Die Zahl *147* setzt sich auf den folgenden Ziffern zusammen:

1, *4* und 7. Jede Ziffer hat einen eigenen Stellenwert, den wir natürlich nicht explizit berechnen müssen, da der Zahlenwert für uns sofort ablesbar ist.

Aber dennoch setzt er sich wie folgt zusammen.

Stellenwert	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
Ziffern	1	4	7

Tabelle 1 - Stellenwertigkeit der einzelnen Ziffern im 10er-System

Hier nun das Ergebnis, wobei ich bei der höherwertigsten Stelle ganz links beginne:

$$\mathbf{1} \cdot 10^2 + \mathbf{4} \cdot 10^1 + \mathbf{7} \cdot 10^0 = 100 + 40 + 7 = \mathbf{147}$$

Unser *10er-System* kennt zehn verschiedene Ziffern, die sich von *0* bis *9* erstecken. Deshalb kommt die Basis *10* zustande. Da wird sich manch einer fragen, warum ich das hier so mache, da das Ergebnis wirklich sofort zu erkennen ist. Stimmt ja auch, doch wenn wir jetzt zum *Binärsystem* wechseln, dann ist diese Erklärung sicherlich hilfreich. Du hast schon gesehen, dass das System, mit dem der Prozessor arbeitet, nur *1* oder *0* kennt und sich somit aus zwei unterschiedlichen Zuständen zusammensetzt. Deshalb arbeiten wir auch mit der Basis *2*. Schauen wir uns dazu natürlich wieder ein passendes Beispiel an. Nehmen wir einmal die Ziffernfolge *10010011*, die wir genauer untersuchen wollen. Du siehst, dass sie lediglich aus die Ziffer *1* bzw. *0* beinhaltet.

Stellenwert	$2^7=128$	$2^6=64$	$2^5=32$	$2^4=16$	$2^3=8$	$2^2=4$	$2^1=2$	$2^0=1$
Ziffern	1	0	0	1	0	0	1	1

Tabelle 2 - Stellenwertigkeit der einzelnen Ziffern im 2er-System

Nun können wir nicht sofort den Wert für unser allseits beliebtes *10er-System* ablesen. Eine Umrechnung wird uns aber helfen, den entsprechenden Wert zu ermitteln.

$$\mathbf{1} \cdot 10^7 + \mathbf{0} \cdot 10^6 + \mathbf{0} \cdot 10^5 + \mathbf{1} \cdot 10^4 + \mathbf{0} \cdot 10^3 + \mathbf{0} \cdot 10^2 + \mathbf{1} \cdot 10^1 + \mathbf{1} \cdot 10^0 = \mathbf{147}$$

Hey, das ist ja genau *die* Zahl, die wir vorher auch schon hatten. Nun siehst Du, wie die Zahl *147* im *2er-System* dargestellt wird. Hier ein nützlicher Hinweis. Wenn Du die Zahl *10010011* siehst, kannst Du auf den ersten Blick nicht erkennen, zu welcher Basis dieser Wert gehört. Es könnte sich ja auch um eine Zahl im *10er-System* handeln. Deshalb gibt es bei der gleichzeitigen Verwendung unterschiedlicher Zahlensysteme einen Zusatz, der rechts angefügt wird und auf die Basis hinweist. Die gerade verwendete Binärzahl würde demnach wie folgt geschrieben.

10010011_2

Ich habe nicht ohne Grund eine *8*-stellige Zahl verwendet, denn der Prozessor kann intern Wert mit einer Breite von *8 Bits* speichern. *Huuh*, da haben wir einen neuen Ausdruck, der einer Erklärung bedarf. Ein *Bit* wird in der Informatik als eine Bezeichnung für eine einzelne Binärziffer verwendet, die den Wert *1* bzw. *0* annehmen kann. Es handelt sich also um die kleinste

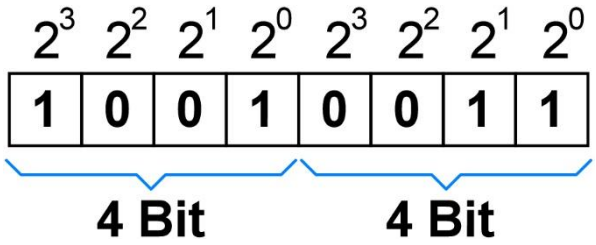
darstellbare Informationseinheit. Wo wir schon bei neuen Begriffen sind, wollen wir auch gleich das *Byte* erklären, das eine Gruppe von *8 Bits* bezeichnet. Eben diese Gruppe hast Du mit der Ziffernfolge *10010011* gesehen. Der Prozessor ist also in der Lage *Byte-Werte* verarbeiten. Wie groß können aber diese Werte maximal sein. Nun, wenn an jeder Stelle eine *1* steht, dann ist das Maximum erreicht.

Anzahl der möglichen Kombinationen: $2^{Anzahl\ der\ Bits} = 2^8 = 256$

Das bedeutet, dass der maximal darstellbare Wert mit *8-Bit* jedoch *255* ist, da die *0* natürlich mitgerechnet wird. Die binäre Schreibweise wird uns noch zu Genüge verfolgen, denn alle Operationen, die durch den Prozessor getätigt werden, laufen auf dieser Ebene ab. Für uns Menschen ist es jedoch recht schwierig, sich mit den Kolonnen von Einsen und Nullen auseinanderzusetzen. Für das Verständnis der internen Abläufe ist es unbedingt notwendig, sich mit diesen elementaren Dingen zu befassen. Wenn es aber um die Schreibweise bzw. das Lesen geht, dann ist es recht umständlich und sicherlich auch unübersichtlich, derartige Ziffernkombinationen zu verwenden. Aus diesem Grunde wurde ein weiteres Zahlensystem entwickelt, dem eine andere Basis zugrunde liegt. *Boah*, schon wieder was Neues? Nein, ganz so schlimm, wie es auf den ersten Blick scheint, wird es nicht. Du wirst nun das *Hexadezimal-System* kennenlernen, dessen Basis die *16* ist.

Wenn ich das richtig sehe, dann haben wir es mit *16* möglichen Kombinationen pro Wertestelle zu tun. Uns stehen aber nur die Ziffern von *0* bis *9* zur Verfügung. Was machen wir denn mit dem Rest?

Gut erkannt! Zum Glück stehen uns weitere Zeichen zur Verfügung, die wir aus unserem Alphabet leihen. Es fehlen also *6* weitere Zeichen, was dazu führt, dass wird die Buchstaben von ***A*** bis ***F*** benötigen. Es hört sich alles viel schlimmer an, als es in Wirklichkeit ist. Wie aber werden *hexadezimale* Zahlen dargestellt? Das ist recht simpel und wir kommen noch einmal zu den *Binärzahlen* zurück, die uns in diesem Fall behilflich sind. Nehmen wir wieder unseren dezimalen Wert *147*, der in binärer Schreibweise ja die folgende Bitkombination *10010011* vorweist. Jetzt werden jeweils *4* Bits zu einer Gruppe zusammengefasst, denn $2^4 = 16$ und eben diese Anzahl der möglichen Kombinationen ist die des hexadezimalen Systems. Schau her:

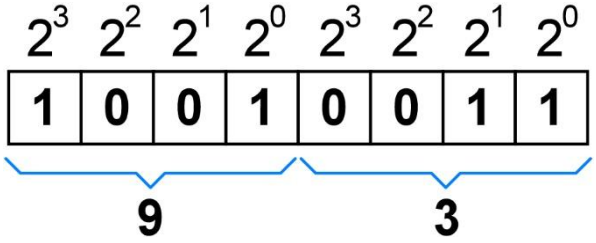


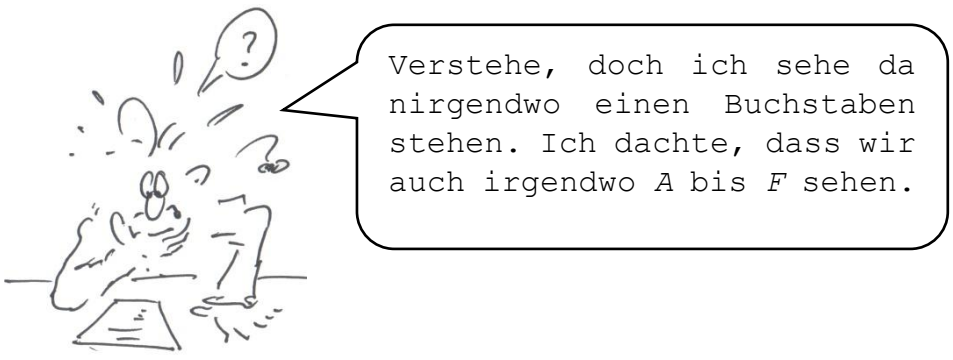
Jeder dieser beiden *4-Bit Gruppen* wird nun eine Ziffer bzw. ein Zeichen aus dem *hexadezimalen System* zugewiesen. Wie das funktioniert, das sehen wir anhand der folgenden Tabelle, die Du - so weit wie möglich - schon auswendig lernen solltest.

Dezimal	Binär	Hexadezimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

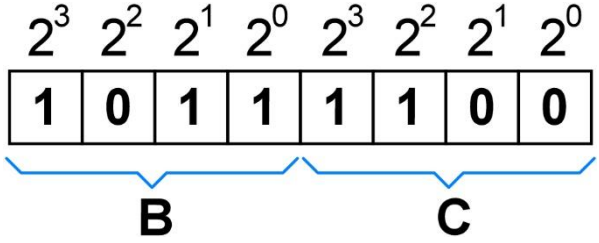
Tabelle 3 - Unterschiedliche Zahlensysteme

Aufgrund dieser Tabelle wollen wir mal schauen, wie sich denn unsere Binärzahl *10010011* im Gewand der hexadezimalen Zahl darstellt.





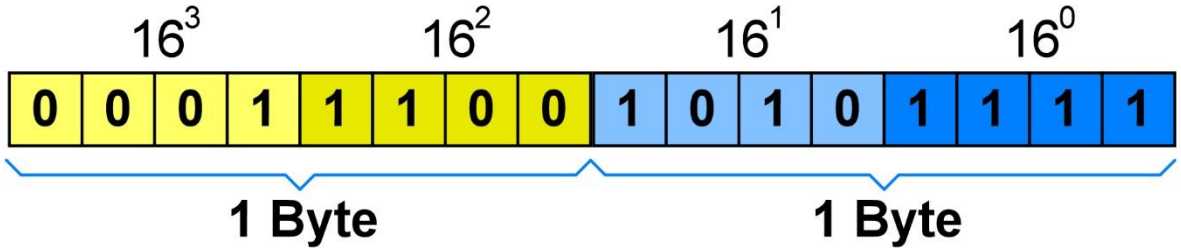
Das muss doch nicht sein! Wenn die einzelnen Stellenwerte nicht größer wie *9* sind, dann benötigen wir auch keinen Buchstaben. Aber schau her:



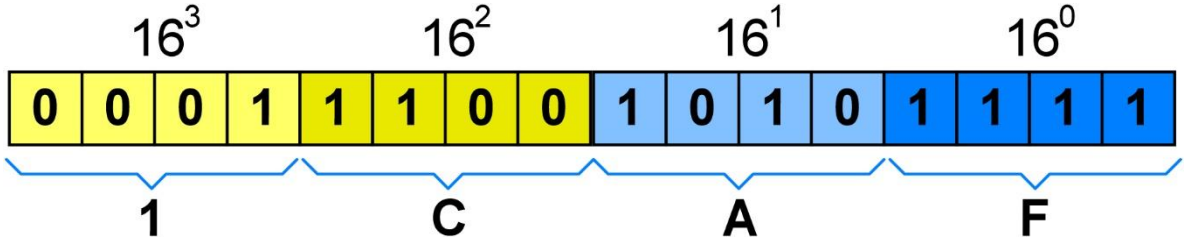
Hier hast Du deine Buchstaben. Die Umrechnung in unser *Dezimalsystem* verläuft ähnlich, wie Du es bei den *Binärzahlen* gesehen hast. Schau her:

$$\mathbf{B} \cdot 16^1 + \mathbf{C} \cdot 16^0 = \mathbf{11} \cdot 16^1 + \mathbf{12} \cdot 16^0 = \mathbf{188}$$

Ist überhaupt nicht so schwer, oder!? Wenn wir jetzt diese Schreibweise noch ein wenig erweitern, dann haben wir die Grundlage, um uns gleich der Programmierung in Maschinensprache zu widmen. Wir haben es beim Prozessor nämlich nicht nur mit *8-Bit* (*1 Byte*) breiten Werten zu tun, sondern auch mit *16-Bits* (*2 Byte*, was auch *Wort* genannt wird). Warum das so ist, darauf kommen wir gleich zu sprechen, wenn wir uns im Groben die Architektur des *6502*-Prozessors anschauen. Doch zunächst die erweiterte Darstellung mit *2 Bytes*.



Wie Du siehst, habe ich jetzt jeder einzelnen *4*-Bit-Gruppe, die sich *1 Nibble* nennt, eine eigene Stellenwertigkeit zugewiesen, die die Basis *16* hat. Du musst für diesen Fall nicht mehr auf Bit-Ebene die Basis *2* verwenden. Dann wollen wir mal schauen, wie wir diesen *16*-Bit-Wert in eine Dezimalzahl wandeln. Wir gehen einfach wieder nach Schema-F vor, was wirklich nicht schwer ist.



$$\mathbf{1} \cdot 16^3 + \mathbf{C} \cdot 16^2 + \mathbf{A} \cdot 16^1 + \mathbf{F} \cdot 16^0 = \mathbf{1} \cdot 16^3 + \mathbf{12} \cdot 16^2 + \mathbf{10} \cdot 16^1 + \mathbf{15} \cdot 16^0 = \mathbf{7343}$$

Das Spielchen könnten wir natürlich endlos weitertreiben, doch wir sind jetzt so weit, dass wir uns im nächsten Kapitel der

internen Struktur des *6502-/6510 Prozessors* zuwenden können. Du hast jetzt *8-Bit* und *16-Bit* breite Werte kennengelernt, was für uns vollkommen ausreichend ist. Auf der nachfolgenden Abbildung ist die Steigerung von einem einzelnen Bit einmal aufgelistet.

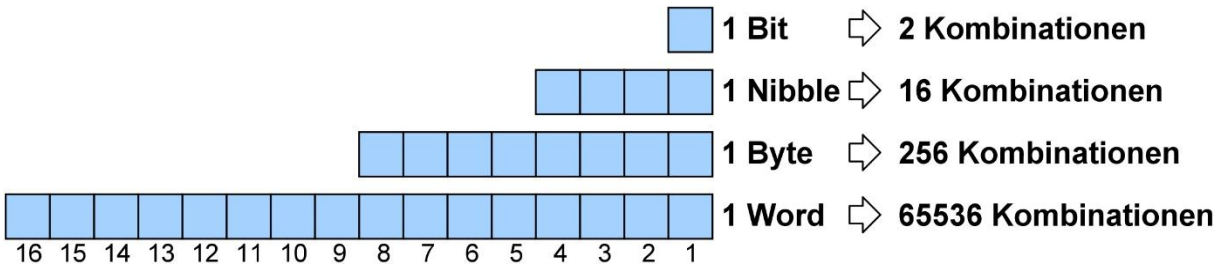


Abbildung 5 - Ein Bit - ein Nibble - ein Byte - Ein Wort

Natürlich können noch wie größere Datenmengen aus mehreren Byte erstellt werden. Meistens geht es dann wieder um eine Verdopplung - so wie immer im Leben - der zugrundeliegenden Menge. Also 2 Bytes (auch Word - Wort genannt), *4* Bytes, *8* Bytes, *16* Bytes, usw. Doch für die Grundlagen zum Verständnis der Digitaltechnik ist es wichtig, erst einmal mit einem einzigen Byte, also 8 Bits, zurechtzukommen. Alles Weitere ist eine einfache Skalierung in höhere Dimensionen, was kein Problem darstellen sollte. Es ist bei 8 Bits wichtig zu verstehen, dass es *256* unterschiedliche Bitkombinationen gibt, die im Bereich von *0* bis *255* angesiedelt sind - nicht von *0* bis *256*! Hinsichtlich der einzelnen Bits sollte noch einmal ein Augenmerk auf den Index gelegt werden, denn ansonsten kommt es unweigerlich zu Problemen. Bit 1 ist nicht das erste Bit, denn die Index-Nummerierung beginnt - wie schon erwähnt - mit der Ziffer *0*. Somit ist die Positionierung der einzelnen Bits wie folgt zu sehen.

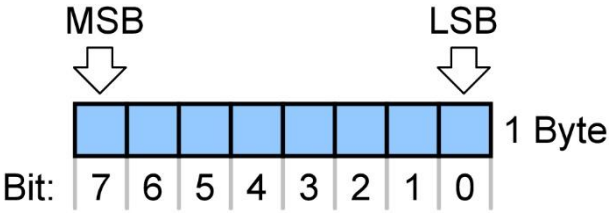


Abbildung 6 - Ein Byte

Es ist zu sehen, dass der Index von rechts bei *0* beginnt und nach links bei *7* endet. Das niederwertige Bit wird *LSB* (Least Significant Bit) und das höherwertige Bit wird *MSB* (Most Significant Bit) genannt. Diese Bezeichnungen sind wichtig zu wissen, denn sie treten in der Literatur immer wieder in Erscheinung. Üblicherweise werden heutzutage die Speicherkapazitäten im Computerumfeld in Potenzen von Byte angegeben. Das Ergebnis ist dann Kilobyte, Megabyte, Gigabyte oder auch Terabyte. Hinsichtlich dieser Bezeichnungen kommt es immer wieder zu Missverständnissen. Warum ist das so? Die Verwendung der Bezeichnungen Kilobyte, Megabyte, Gigabyte,

Terabyte ist teilweise etwas unterschiedlich. Nimmt man es genau, dann sind die Zehnerpotenzen der Bytes gemeint. Ein Kilobyte sind demnach *1.000* einzelne Bytes. Richtigerweise wird das Kilobyte aber aus einer Zweierpotenz berechnet und zwei hoch zehn entspricht in diesem Fall nicht *1.000* sondern *1.024* einzelnen Bytes. Ist das verständlich? Vielleicht nicht so ganz! Deswegen hier noch ein paar weitere Erläuterungen, warum *1.000* nicht *1.024* ist. Die Bezeichnungen *Kilo*, *Mega*, *Giga* und *Tera* sind ein Vielfaches von *10* hoch *3*, was *1.000* entspricht.

Die Potenzen sind in der folgenden Tabelle zu sehen.

Potenzen	Anzahl Bits	Bezeichnung	Abkürzung
10^3	1.000	Kilobit	kBit
10^6	1.000.000	Megabit	MBit
10^9	1.000.000.000	Gigabit	GBit
10^{12}	1.000.000.000.000	Terabit	TBit

Tabelle 4 - Unterschiedliche Potenzen

In der Informatik werden diese Werte aber ebenfalls in einem Atemzug für ein Vielfaches von *2* hoch *10*, was *1.024* entspricht, verwendet. Hinsichtlich eines Kilobits ist also zwischen einem dezimalen und einem binären Kilobit zu differenzieren. Das dezimale Kilobit besitzt *1.000* Bit und das binäre Kilobit etwas mehr, also *1.024* Bit. Wir müssen realisieren, dass es Unterschiede in den Zahlensystemen gibt, die der Mensch verwendet und die im Computer zur Anwendung kommen. Abschließend zu dieser kurzen Einführung zeige ich auf der folgenden Abbildung die Zahlensysteme untereinander.

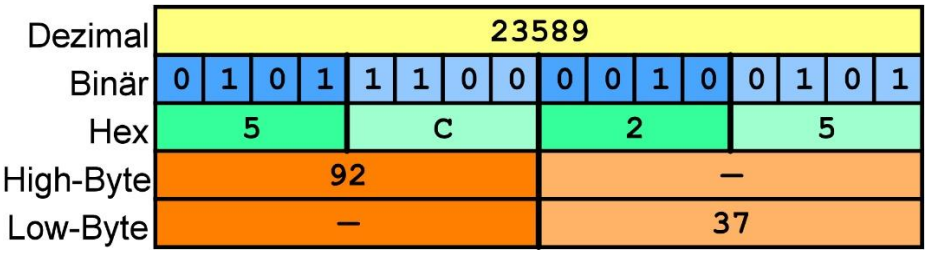


Abbildung 7 - Verschiedene Zahlensysteme auf einen Blick

Es ist hier sehr schön zu erkennen, dass eine Aufteilung der *16*-Bit Binärzahl in Nibbles, also *4*-Bit-Gruppen, direkt zu den entsprechenden hexadezimalen Werten führt. Da wir es mit einem *8*-Bit-Computer zu tun haben, aber ein *16*-Bit Adressbus vorhanden ist, müssen in den *8*-Bit breiten Speicherstellen auch Adresse hinterlegt werden. Aus diesem Grund werden diese in zwei Byte-Gruppen je *8* Bits aufgeteilt. Das linke Byte repräsentiert das sogenannte ***High-Byte***, also höherwertige Byte, das rechte das ***Low-Byte***, das niederwertige Byte.

Zur Bestimmung der Werte High- bzw. Low-Werte kann man recht einfach vorgehen.

Umrechnung von Dezimal in Low-/High-Byte

Ich führe nachfolgend eine Ganzzahl-Division durch.

$$High\ Byte = \frac{23589}{256} = 92$$

Zur Berechnung des Low-Bytes muss der Rest der Division berücksichtigt werden.

$$Low\ Byte = 23589 - 92 \cdot 256 = 37$$

Umrechnung Low-/High-Byte in Dezimal

Um auf den Dezimalwert zu kommen, wenn Low-/High-Byte vorliegen, müssen das *Low-Byte* und das *High-Byte x 256* addiert werden. Das macht in unserem Fall.

$$Dezimalwert = 37 + 92 \cdot 256 = 23589$$

Ein sehr wichtiger Aspekt bei der Ablage der *16*-Bit-Werte im Speicher ist der Umstand, dass immer zuerst das *Low-Byte* gefolgt vom *High-Byte* im Speicher zu finden sind. Ich komme natürlich im Detail noch darauf zu sprechen.

Der 6502-/6510-Prozessor und das Bussystem

Ein Prozessor, der das Herzstück jedes Computers ist, benötigt natürlich zum Arbeiten irgendwelche Möglichkeiten der Kommunikation mit der Außenwelt. Seine Berechnungen will er nicht unbedingt für sich behalten, sondern an die unterschiedlichsten Stellen weiterreichen. denn der Prozessor verfügt im Grunde genommen über keinen eigenen Speicher. Das stimmt jedoch nicht ganz. Später mehr dazu. Zu diesem Zweck gibt es die unterschiedlichsten *Bus-Systeme*. Schauen wir uns dazu die folgende Grafik ein wenig genauer an. Sie beschränkt sich auf das Wesentlichste und enthält nicht alle vorhandenen Details, was uns aber in diesem Fall und im jetzigen Moment nicht weiter stören sollte.

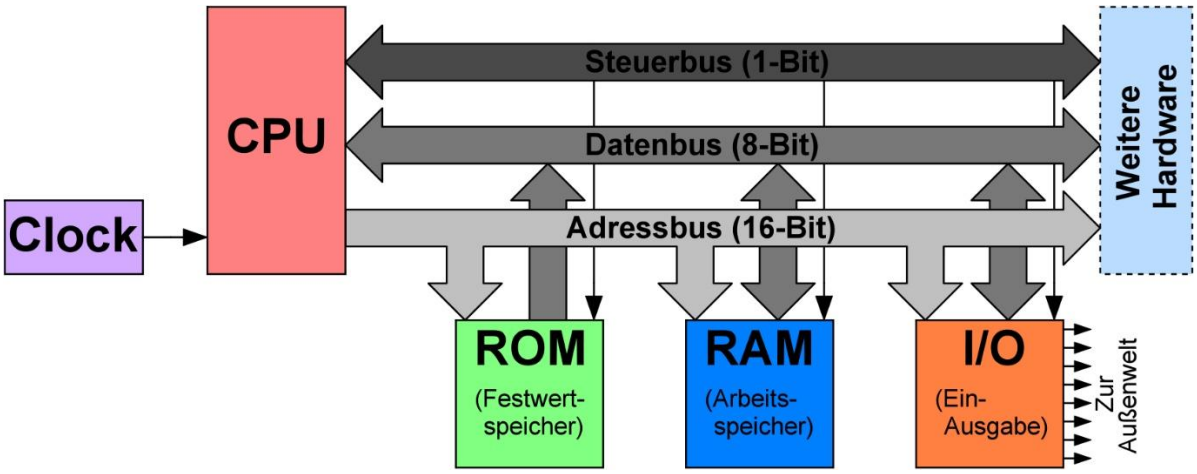


Abbildung 8 - Die Grundstruktur eines Computersystems

In der linken oberen Ecke befindet sich unser Hauptakteur, der *6502-/6510-Prozessor*. Er ist über mehrere elektrische Leitungen mit verschiedenen elektronischen Komponenten verbunden. Der Taktgeber - auch *Clock* genannt - gibt die Taktfrequenz beziehungsweise Geschwindigkeit des Prozessors vor, mit der gearbeitet wird. Je höher ein derartiger Takt ist, desto schneller ist die Verarbeitungsgeschwindigkeit. Im Idealfall schaut ein derartiger Takt in Form von Rechtecksignalen wie folgt aus.

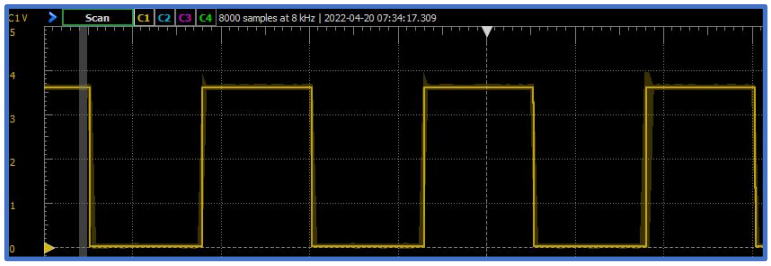


Abbildung 9 - Der Taktimpuls

Ein *Bus* besteht aus mehreren elektrischen Leitungen und dient der Kommunikation zwischen mehreren Teilnehmern, die einen gemeinsamen Übertragungsweg nutzen. Du siehst, dass wir es hier mit drei unterschiedlichen Bus-Systemen mit unterschiedlicher Datenbreite zu tun haben.

- Steuerbus (*1*-Bit Datenbreite)
- *Daten-Bus* (*8*-Bit Datenbreite)
- *Adress-Bus* (*16*-Bit Datenbreite)

Der *Steuerbus*, der u.a. die Schreib-/Lese-Zugriff regelt und die einzelnen Bus-Teilnehmer anspricht, ist erst einmal zweitrangig. Der *Adress-Bus* stellt quasi eine *Einbahnstraße* dar, denn es werden nur Adressen vom Prozessor zu angeschlossenen Bus-Teilnehmern verschickt, um dort Daten abzurufen. Im Gegensatz dazu kann der *Daten-Bus* in beiden Richtungen betrieben werden.

Daten werden sowohl vom Prozessor verschickt, als auch empfangen.

In einem Computer gibt es zur Verwaltung der Daten die unterschiedlichsten Speicherbausteine wie z.B. das *RAM* (***R**andom **A**ccess **M**emory*) oder das *ROM* (***R**ead **O**nly **M**emory*). Der *Arbeitsspeicher*, in dem Du z.B. Deine Programme ablegen kannst, befindet sich im *RAM*. Es handelt sich um einen *Schreib-/Lese-Speicher*, der flüchtig ist, was bedeutet, dass er seine Informationen nach Verlust der Betriebsspannung verliert. Des Weiteren gibt es noch das *ROM*, wie sich System-Routinen oder auch das Betriebssystem befinden. Dieser *Nur-Lese-Speicher* behält natürlich nach dem Abschalten des Computers seine Informationen. Dann gibt es noch Bausteine, die eine Kommunikation mit einer ggf. angeschlossenen Peripherie wie z.B. *Floppy-Drive*, *Kassettenlaufwerk* oder auch *Drucker* ermöglichen. Alles das wird über die unterschiedlichen Bus-Systeme geregelt, die die Daten intern und extern verschieben. Der Grund, warum der *Adress-Bus* im Gegensatz zum *Daten-Bus* mit *16-Bit* ausgestattet wurde hat natürlich den Grund der erweiterten Adressierung. Ein Speicherbereich mit *256* Adressen wäre etwas sehr knapp bemessen. Mit *16-Bit* sieht die Sache schon ganz anders aus.

Anzahl der möglichen Kombinationen: $2^{Anzahl\ der\ Bits} = 2^{16} = 65536$

Wenn wir Zahl *65.536* in Form von Bytes angeben, dann sollten wir die nächst höhere Maßeinheit verwenden. Das *Kilobyte*. Das sind jetzt aber nicht *1.000 Bytes*, sondern *1.024 Bytes*. Warum? Ich sprach das Thema schon einmal an. $2^{10} = 1.024$. Aus *65.536 Bytes* werden dann *64 Kilobytes* (*KB*).

Schauen wir uns doch einmal einen Zugriff auf einen Speicherbaustein genauer an. Wie läuft das ganze ab? Der Prozessor möchte die Daten von einem Speicherbaustein abrufen, um mit ihm zu rechnen. Dazu muss er erst einmal wissen, wo sich an welcher Stelle diese Daten im Speicher befinden. Ein Speicher ist über sogenannte *Adressen* organisiert. Ganz ähnlich einem Schrank, der über mehrere Schubladen verfügt, die jeweils mit einem Schildchen versehen sind, wo sich fortlaufende Nummern drauf befinden.

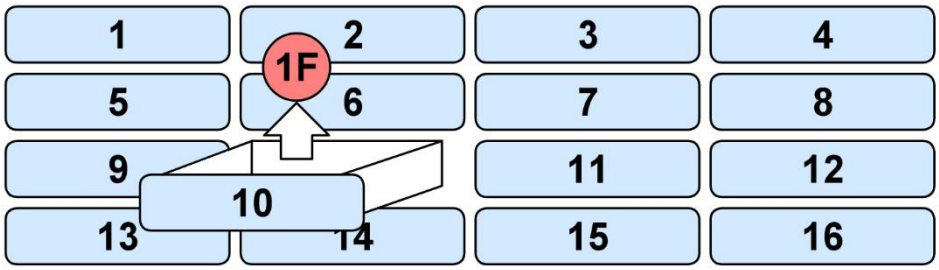
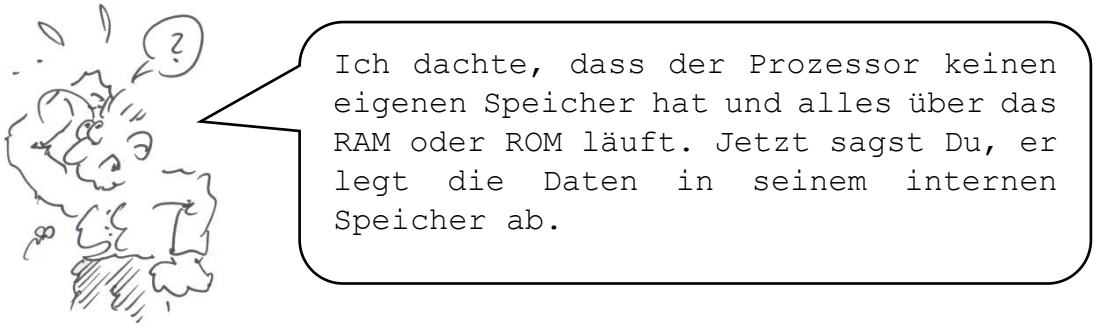


Abbildung 10 - Eine Schublade mit darin enthaltenen Informationen wird geöffnet

In der Schublade mit der *Adresse 10* befindet sich ein Inhalt - hier *1F*, der uns interessiert. Wie läuft das aber für den Prozessor ab? Folgende Schritte werden durchgeführt:

- An welcher *Adresse* im Speicher befinden sich die Daten?
- Ablage der erforderlichen Adresse auf dem *Adressbus*
- Abrufen der Daten aus der benannten Speicherstelle und Ablage auf dem *Datenbus*
- Einlesen Der Daten in den internen Speicher des Prozessors
- Die Bearbeitung der Daten kann beginnen



Ok, das konntest Du nicht wissen. Der *6502-/6510-Prozessor* verfügt über drei Speicher, die er für seine internen Berechnungen verwendet, das ist aus heutiger Sicht nicht viel, doch es reicht. Diese Speicherbereiche werden übrigens *Register* genannt. Du wirst sie gleich kennenlernen.

Die Register des Prozessors

Damit der Prozessor schnell seine Berechnungen durchführen kann und nicht zum Arbeiten immer auf externen Speicher zurückgreifen muss, was zusätzlichen (Zeit-)Aufwand bedeutet, nutzt er seine internen Speicherbereiche (*schneller Speicher*), die auch *Register* genannt werden.

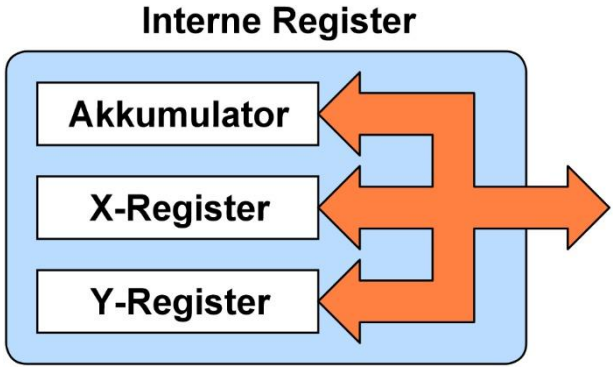


Abbildung 11 - Die internen Register des 6502 (Teil 1)

Der *Akkumulator* - kurz ***Akku*** genannt - ist dabei einer *der* Speicher, der am häufigsten genutzt wird. Kein Weg führt eigentlich an ihm vorbei, denn die meisten Rechenoperationen

erfolgen über ihn, wobei sich die Ergebnisse dann wiederum darin befinden. Das *X-* bzw. *Y-Register* kann nicht zum Rechnen verwendet werden und wird meistens dafür genutzt, um Werte rauf- bzw. runter zuzählen. Diese Operationen werden *Inkrementieren* bzw. *Dekrementieren* genannt.

Ich weiß, es ist vielleicht noch etwas früh, doch kannst Du mir nicht schon mal einen Befehl zeigen, um einen Wert in den Akkumulator zu laden.

Du hast Recht, das sollte ich tun. Das ist eigentlich ein guter Zeitpunkt, um Dir den Unterschied zwischen Maschinensprache und *dem* zu zeigen, worin wir gleich programmieren werden.

Aber ich dachte, dass wir in Maschinensprache programmieren werden. Was soll das denn nun wieder bedeuten?

Die *Maschinensprache* besteht lediglich aus *Zahlen* bzw. *Codes*. Wir programmieren zwar gleich in *Maschinensprache*, nutzen dazu aber sogenannte *Assembler-Befehle*. Das sind kurze Abkürzungen von englischen Befehlen, die sich *Menmonics* nennen. Um z.B. einen Wert in den Akkumulator zu laden, wird der Befehl ***L**oa**DA**kkumulator* verwendet. Na ja nicht ganz. Die *Mnemonics* setzen sich immer aus *drei Buchstaben* zusammen, so dass der entsprechende Befehl *LDA* lautet. Um also einen Wert in den *Akku* zu laden, kannst Du den folgenden Befehl verwenden.

LDA #$1F

Natürlich musst Du dem Befehl *LDA* mitteilen, welchen *Wert* Du in den Akku laden möchtest. Das erfolgt über den nachfolgenden Wert *#$1F*. Wir schauen uns das kurz im Assemblerprogramm *SMON*, welches von mir durchgehend Verwendung findet. Sehen wir uns das gleich genauer an. Doch zuvor möchte ich die Informationen über die internen Register des Prozessors noch vervollständigen, denn die drei von mir genannten (Akkumulator, X-Register u. Y-Register) waren nur die halbe Miete.

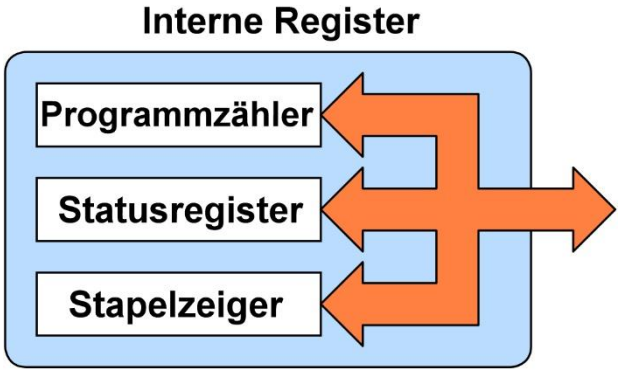


Abbildung 12 - Die internen Register des 6502 (Teil 2)

Der *Programmzähler* - auch Programmcounter (PC) genannt - ist ein 16-Bit-Register. Sein Inhalt bestimmt, aus welcher Speicherstelle der ***nächste*** abzuarbeitende Befehl geholt wird. Dieses Register wird vom Prozessor selber verwaltet und man hat damit in der Regel nichts zu tun. Dennoch ist es natürlich überaus wichtig, die Funktion dieses Registers zu verstehen.

Da der *6502/6510* über einen *16*-Bit Adressraum (*64* KByte) verfügt, muss ein derartiger Zähler, der in der Lage sein soll, alle möglichen Adressen anzusprechen, ebenfalls *16* Bit breit sein. Der *PC* merkt sich also, wo wir uns gerade im laufenden Programm befinden und was als nächstes dran ist und zeigt auf die Adresse des Befehls, der als nächstes auszuführen ist. Der *PC* ist in zwei *8*-Bit-Register für das *Low*- bzw. *High-Byte* anstelle eines einzigen *16*-Bit Registers organisiert und er ist deshalb so strukturiert, weil einige Adressierungsmodi das Low-Byte separat behandeln.

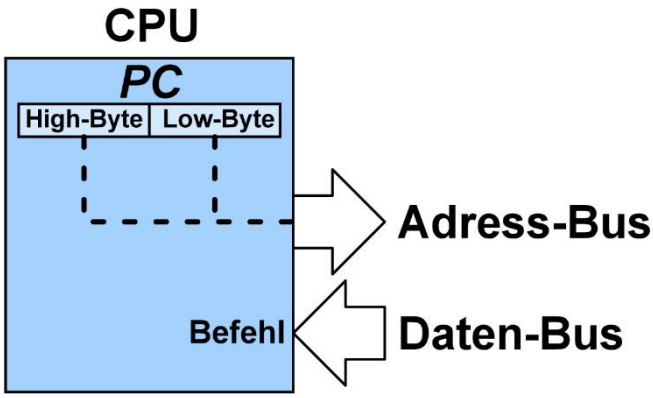


Abbildung 13 - Der Programmzähler

Die Funktionsweise des *PC* ist ziemlich einfach: Er wird mit einem bestimmten initialen Wert geladen und erhöht diesen Wert dann automatisch im Laufe des Programms. Diese Aufgabe wird durch einen einfachen Inkrementierer erledigt, was eine schrittweise Erhöhung eines Wertes bedeutet. Da es jedoch unterschiedlich lange Befehle (*1* Byte, *2* Byte oder *3* Byte) gibt, muss das Erhöhen des Programmzählers entsprechend angepasst werden, was durch den Zyklus/Opcode-Decoder realisiert wird. Natürlich zählt der

Programmzähler in Abhängigkeit des Programms beziehungsweise der Befehle nicht einfach nur aufwärts. Es gibt auch Sprungbefehle, die fordern, dass zu einer bestimmten Stelle des Speichers gesprungen werden muss. Es gibt da zum Beispiel einen *JMP*-Befehl (JMP=Jump, Sprung), der den Programmzähler mit der angegebenen Adresse versieht. Es erfolgt dann im nächsten Schritt ein Sprung zur genannten Adresse. Auf diese Weise können Verzweigungen oder Schleifen in einem Programm ermöglicht werden. Ich möchte die Arbeitsweise der CPU an zwei Szenarien verdeutlichen.

Der Lesevorgang aus einem ROM

Um die Informationen aus einem Speicher abzurufen, müssen nacheinander unterschiedliche Schritte abgearbeitet werden.

1. Erforderliche Adresse auf den Adressbus legen
2. Aktivierung der Steuerleitung für das Lesen aus dem Speicherbaustein
3. Die angesprochene Speicheradresse legt automatisch ihren Inhalt auf den Datenbus
4. Die CPU übernimmt die Daten vom Datenbus und deaktiviert die Steuerleitung für das Lesen aus dem Speicher

Der Schreibvorgang in ein RAM

Um Informationen in einen Speicher durchzuführen, muss natürlich sichergestellt werden, dass dieser Speicherbaustein auch beschreibbar ist, also zum Beispiel ein RAM-Baustein, denn in ein ROM könnte niemals nach der eigentlichen Programmierung ab Werk wieder etwas geschrieben werden.

1. Erforderliche Adresse auf den Adressbus legen
2. Die CPU legt die zu speichernden Daten auf den Datenbus
3. Aktivierung der Steuerleitung für das Schreiben in den Speicherbaustein
4. Die angesprochene Speicheradresse des Speicherbausteins nimmt die Informationen des Datenbusses auf und speichert sie
5. Deaktivierung der Steuerleitung für das Schreiben in den Speicherbaustein

Der *Stapelzeiger* - auch Stackpointer (SP) genannt - zeigt auf den sogenannten Stapelspeicher oder kurz Stack, der unter anderem für Unterprogramme oder für temporäre Speicherung von Daten benutzt wird.

Das Statusregister gibt Auskunft über das Ergebnis des zuletzt ausgeführten Befehls und ist Grundlage für Entscheidungen und bedingte Befehle. Von den acht zur Verfügung stehenden Bits des Statusregisters werden sieben Bits als sogenannte *Flags* benutzt. Flags oder auch Flaggen sind über bestimmte Befehle direkt abfragbar. So gibt es zum Beispiel Sprungbefehle, die nur dann

zur Ausführung kommen, wenn ein solches Flag gesetzt oder nicht gesetzt ist. Diese Flags können in der Interpretation wahr (=*1*) oder falsch (=*0*) sein und signalisieren darüber eine vorherrschende Bedingung. Das Statusregister ist folgendermaßen aufgebaut.

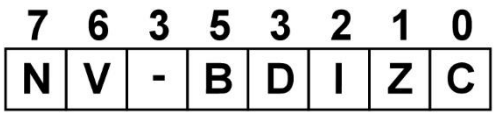


Abbildung 14 - Das Statusregister

Ich möchte an dieser Stelle nur kurz auf deren Bedeutungen eingehen und die Details für später aufheben, wenn wir konkret Beispiele bearbeiten. Ich gehe von rechts nach links, also von Bit *0* zu Bit *7* vor.

C - (*Carry*) Das Carryflag zeigt an, ob bei einer Operation ein Überlauf aufgetreten ist. Das tritt z.B. dann auf, wenn beim Addieren zweier Werte das Ergebnis größer als *255* wird und es sich nicht mehr mit 8 Bit darstellen lässt. In diesem Fall wird das Carry-Flag gesetzt.

Z - (Zero) Das Zero- oder Nullflag wird immer dann gesetzt, wenn das Ergebnis einer Operation Null ist.

I - (Interrupt Disable) Dieses Flag entscheidet, ob Interrupts im Programm erlaubt sind.

D - (Decimal) Das Dezimalflag bestimmt, ob Addition oder Subtraktion im Dezimalmodus durchgeführt wird.

B - (Break) Das Breakflag zeigt eine Unterbrechung durch den BRK-Befehl an.

V - (Overflow) Das V-Flag wird nur benötigt, wenn mit vorzeichenbehafteten Zahlen gerechnet wird und zeigt dann einen Überlauf an.

N - (Negative) Dieses Flag wird immer dann gesetzt, wenn das Ergebnis einer Operation einen Wert von größer als *127* ergibt, wobei das siebente Bit gesetzt ist. Die Bezeichnung Negative kommt daher, dass man Werte größer *127* als negative Zahlen interpretieren kann. Abschließend zu dieser kurzen Einführung hier die Auflistung der Register in Gänze.

Das Statusregister

Der Inhalt des Statusregisters bleibt solange unverändert, bis eine neue Operation erfolgt, die Auswirkungen auf das Register hat.

Bei jedem neuen Befehl werde ich eine kleine Übersicht zeigen, ob und welche Flags im Statusregister nach der Ausführung beeinflusst werden können. Das schaut dann zum Beispiel wie folgt aus.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	✓

Überall dort, wo ein Check-Zeichen zu sehen ist, können sich Änderungen ergeben, die von der Ausführung des jeweiligen Befehls abhängen. Im gezeigten Beispiel können also folgende Flags betroffen sein.

- Negative-Flag
- Zero-Flag
- Carry-Flag

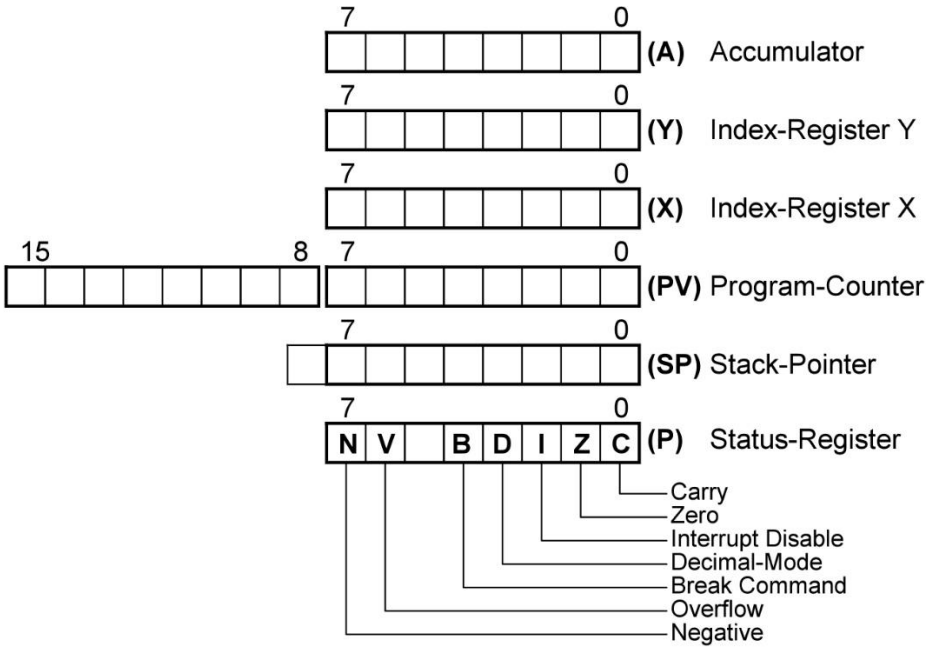


Abbildung 15 - Die Register der 6502 CPU

Befehle und Daten

Wenn es um die unterschiedlichen Informationen geht, die über einen Datenbus versendet werden, dann können wir diese in zwei Kategorien unterteilen. Angenommen, es soll ein bestimmter Wert in ein internes Register geladen werden. Da es aber unterschiedliche Formen von CPU-Register gibt, muss ein derartiger Vorgang mit einem eindeutigen Befehl verbunden sein. Anschließend folgt dann der zu speichernde Wert in Form von Daten. Da haben wir schon die beiden Kategorien: *Befehle* und *Daten*. Man kann jedoch nicht von vornherein festlegen, was Befehle und was Daten sind. Da in einem Computer alles mit Zahlen abläuft, ist dem reinen numerischen Wert *15* nun nicht anzusehen, ob es sich um einen Befehl oder um Daten handelt. Prinzipiell

besteht also kein Unterschied zwischen Befehlen und Daten. Woher weiß aber die CPU, was nun ein Befehl ist und was dann die Daten sind? In der Regel läuft das aber meistens in der folgenden Reihenfolge ab: Was (Befehl) soll womit (Daten) getan werden? Natürlich kann das nicht verallgemeinert werden, denn es gibt im Speicher auch Bereiche, die lediglich mit einer Vielzahl von Daten angehäuft sind. Dazu später aber mehr.

Ganz zu Beginn möchte ich eine Internetseite vorstellen, die mit sehr vielen Informationen hinsichtlich des Befehlssatzes der *6502-/6510*-CPU aufwartet.

Das 6502 Instruction Set

https://www.masswerk.at/6502/6502_instruction_set.html

SMON - Erste Runde

- Das Laden von SMON
- Das Starten von SMON
- Den Assembler nutzen
- Die CPU-Register in SMON
- Das Starten von Maschinenprogrammen
- Das Speichern von Maschineprogrammen

SMON - Erste Runde

Einige Computer besitzen ein eingebautes Monitorprogramm, das *MLM* (***M**achine **L**anguage **M**onitor*) genannt wird. Wenn wir uns an den PET/CBM oder CBM3032 erinnern, so kann man dort über einen entsprechenden SYS-Befehl in den MLM einsteigen.

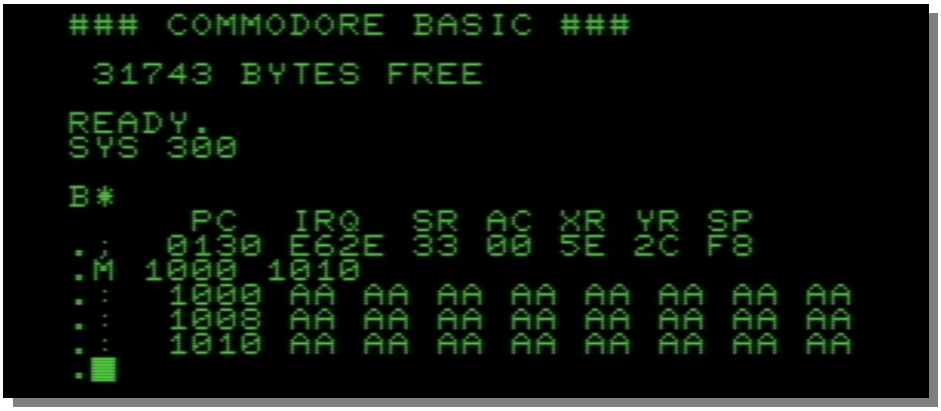


Abbildung 16 - Das Monitorprogramm im CBM3032

Der C64 besitzt kein derartiges Monitor-Programm und somit muss dieses in den Speicher geladen werden, um es nutzen zu können. Der Assembler *SMON* ist ein beliebter Maschinensprachemonitor für den C64, der von *Norfried Mann* und *Dietrich Weineck* entwickelt wurde. Beim SMON handelt es sich um ein reines Monitor-Programm, das direkt im Speicher arbeitet. Man könnte das Programm deswegen auch *Direkt-Assembler* nennen. Ebenfalls sehr schön ist das Programm *Supermon* von *Jim Butterfield*.

Im Gegensatz dazu arbeitet zum Beispiel der *Turbo-Assembler* (TASM 7.4) - auch ein sehr gutes Programm - etwas anders. Er nimmt eine Textdatei als Grundlage eines zu schreibenden Programms und übersetzt dieses später in zwei Durchgängen Maschinensprache. Es handelt sich um einen sehr leistungsstarken Assembler mit vielen Möglichkeiten. Dieser Ansatz liefert ebenfalls ein sehr nostalgisches Programmiergefühl.

Natürlich hätte ich an dieser Stelle auch das sehr verbreitete *C64-Studio* oder *CBM prg Studio* nutzen können, die in ihren Funktionsumfängen schon faszinierend und sehr beeindruckend sind. Doch ich wollte ein längst vergangenes Gefühl aufkommen lassen und nicht ein anderes Betriebssystem an erster Stelle setzen. Sicherlich gibt es bei der Nutzung von *SMON* schon ein paar Einschränkungen und gerade, wenn es um Labels, also Sprungmarken geht, müssen Abstriche gemacht werden. Es gibt zwar eine Möglichkeit und ich gehe auch darauf ein, doch wie bei einem richtigen Assemblerprogramm ist es leider nicht zu handhaben. Das soll uns nicht weiter stören, denn es geht auch ohne und authentischer als die genannten Studio-Entwicklungsumgebungen ist das mit *SMON* allemal.

Das Laden von SMON

Laden wir SMON in VICE hinein. Doch wo bekommen wir das Programm eigentlich her?

	Wo bekommt man das Assemblerprogramm SMON her?
https://www.n3rdroom.de/download/smon-c64-monitorprogramm/	

Bei der Datei handelt es sich um eine gepackte ZIP-Datei, die nach dem entpacken ein Diskimage mit der Endung *D64* enthält. Im ersten Schritt wird über den Menüpunkt

File > Attach disk image > Drive #8

der Dateibrowser aufgerufen, um zur entpackten Datei

smon.d64

navigieren.

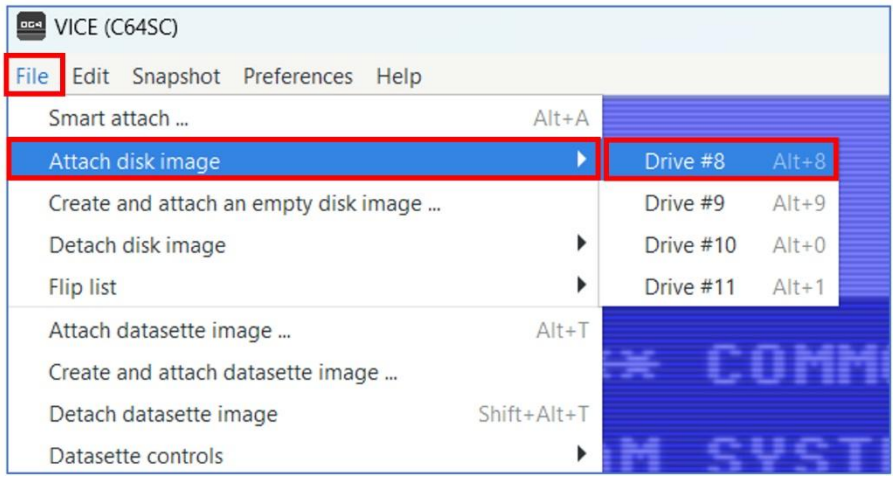


Abbildung 17 - SMON wird eingebunden - Schritt 1

Im zweiten Schritt wird die genannte Datei ausgewählt und auf die *Attach/Load*-Schaltfläche geklickt.

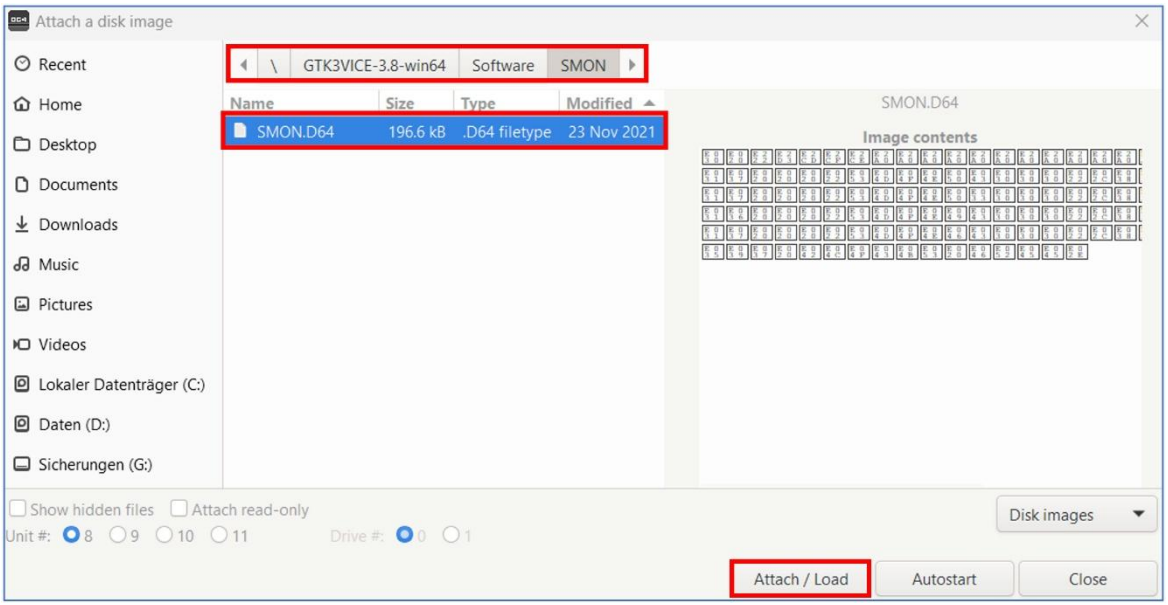


Abbildung 18 - Das Image wird ausgewählt

Im Anschluss können wir uns den Inhalt der eingebundenen Diskette in Laufwerk *8* ansehen.

Abbildung 19 - Der Inhalt der Diskette

Jetzt laden wir mit dem folgenden Befehl SMON in den Speicher und geben anschließend noch *NEW* ein, um eine spätere Fehleranzeige *„OUT OF MEMORY“* zu vermeiden, wenn die Sprache Basic genutzt werden soll.

LOAD "SMONPC000",8,1

NEW

```
LOAD"SMONPC000",8,1

SEARCHING FOR SMONPC000
LOADING
READY.
NEW

READY.
```

Abbildung 20 - SMON wurde ab $C000 in den Speichergeladen

Das Starten von SMON

Nun ist SMON zwar im Speicher ab Adresse *$C000* geladen, doch noch nicht gestartet. Das erfolgt dann über die Eingabe von

SYS 49152

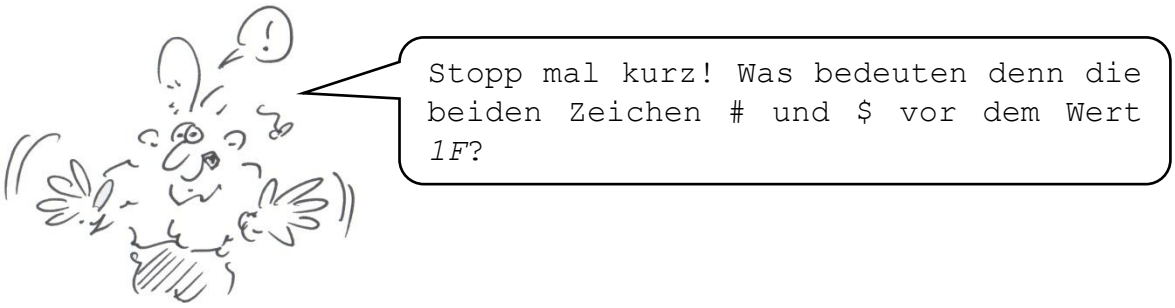
```
SYS 49152

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

Abbildung 21 - Der Monitor wurde aufgerufen

Der *SYS*-Befehl springt übrigens direkt an die angegebene Speicherstelle und führt dort das vorhandene Maschinenprogramm aus. Auf diese Weise können auch Systemroutinen aufgerufen werden. Doch zurück zu Meldung nach dem SYS-Aufruf. Das sieht auf den ersten Blick vielleicht etwas ungewohnt aus, doch wir werden auf die einzelnen Bedeutungen noch zu sprechen kommen. Zu Beginn werden einige Abkürzungen wie *PC*, *SR*, *AC*, usw. mit darunter stehenden Werten ausgegeben. Es handelt sich um die Anzeige wichtiger Statusinformationen. Dazu später mehr.

	Wie wird SMON bedient?
https://www.c64-wiki.de/wiki/SMON	

Eine Eingabezeile beginnt im Monitorprogramm immer mit einem Punkt und es wird Dir darüber signalisiert, dass Du Dich nicht mehr im *BASIC-Interpreter* befindest. Wir wollen jetzt einfach einmal den Akku mit dem hexadezimalen Wert *1F* laden.



Sicher eine berechtigte Frage! Ok, ich fange am besten einmal mit dem *Dollar-Zeichen $* an. Es wird immer dann verwendet, wenn eine Zahl im *hexadezimalen* Format angegeben wird. Und *1F* ist eindeutig eine *hexadezimale* Zahl. Das *Doppelkreuz #* oder auch *Lattenzaun* genannt, bezieht sich auf die *Adressierung*, die hier angewandt wird. Darauf kommen wir im Detail noch zu sprechen.

Den Assembler nutzen

Wir verwenden im Moment eine *unmittelbare Adressierung*. Das bedeutet, dass der Wert *$1F unmittelbar* und ohne Umwege in den *Akku* geladen wird. Folgendes wird in den Monitor eingegeben. Zu Beginn muss erst einmal die Startadresse festgelegt werden, ab der das Assemblerprogramm im Speicher zu liegen kommen soll. Die allgemeine Syntax lautet

Axxxx

- A : Assembliere (Assemble)
- xxxx: Startadresse in hexadezimaler Schreibweise

Ich habe die Startadresse *$4000* gewählt. Die Eingabe lautet also.

A4000

```
  PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F6  10110000
.A4000
 4000
```

Hinter der genannten Adresse blinkt der Cursor und erwartet jetzt einen Befehl. Um den Akku mit dem Wert *#$1F* zu laden, wird der folgende Befehl eingegeben, wobei ich noch nicht mit der *RETURN*-Taste bestätigt habe.

```
  PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F6  10110000
.A4000
 4000 LDA #$1F
```

Erst nach der Bestätigung wird der Befehl vom Assembler erkannt und es erscheinen die entsprechenden OP-Codes. Es ist zudem zu sehen, dass das von mir eingegebene $-Zeichen, was ja für hexadezimale Zahlen steht, nach der Eingabe nicht mehr

auftaucht, weil alle Zahlen als hexadezimale Werte interpretiert werden.

```
  PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F6  10110000
.A4000
 4000  A9 1F     LDA #1F
 4002
```

Die Eingabe des Mnemonics, also des lesbaren Kürzels für einen Befehl in einer Assemblersprache, erscheinen in der Zeile und davor sind die Maschinencodes in hexadezimaler Schreibweise zu sehen.

A9 1F

Das Monitorprogramm hat den eingegebenen Befehl also erkannt und direkt in Maschinensprache übersetzt. Ich komme gleich zur Bedeutung der angezeigten Maschinencodes. Der Cursor wandert in die nächste Zeile und wartet auf den nächsten Befehl. Doch was ist überhaupt ein OP-Code?

Was ist denn ein OP-Code?

Ein *OP-Code*, auch operation code genannt, ist eine Zahl, die die Nummer eines Maschinenbefehls für eine bestimmte CPU repräsentiert. In Summe bilden alle OP-Codes zusammen den Befehlssatz der CPU.

Zudem ist in der darauffolgenden Zeile schon die Startadresse des nächsten Befehls - *4002* - zu sehen, die 2 Byte größer ist, als die zuvor eingegebene Startadresse. Es handelt sich aktuell also um einen *2* Byte-Befehl. Im nächsten Schritt muss der Programmausführung des Maschinenprogramms mitgeteilt werden, dass eine Unterbrechung erfolgen soll, was über den *BRK*-Befehl bewirkt wird. BRK steht für *Break* und stoppt die Ausführung. Es handelt sich hierbei nun um einen *1* Byte-Befehl. Wir geben also BRK ein und bestätigen diese Eingabe wieder mit der *RETURN*-Taste, was zur folgenden Anzeige führt.

```
  PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F6  10110000
.A4000
 4000  A9 1F     LDA #1F
 4002  00        BRK
 4003
```

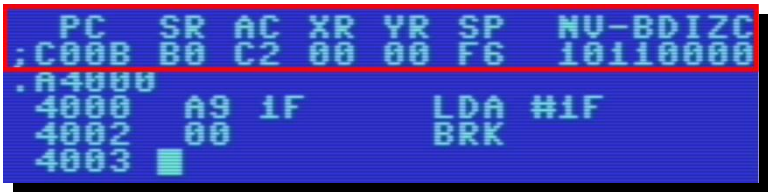
Nun ist alles soweit im Speicher hinterlegt, dass das kleine Programm gestartet werden kann. Doch zuvor ein paar Hinweise zur Anzeige im Monitor, auf die ich noch nicht eingegangen bin.

Eine informative Seite im Internet, die unter anderem eine sehr gute Übersicht aller OP-Codes liefert, ist unter der folgenden Adresse zu finden.

	Wo findet man eine Liste aller OP-Codes?
https://www.pagetable.com/c64ref/6502/?tab=2#	

Die CPU-Register in SMON

Der hier rot umrandete Bereich zeigt wichtige Informationen über den Zustand der CPU und den Inhalt der Register.



Ich möchte mich an dieser Stelle erst einmal auf den Inhalt des Akkus beschränken, der mit *AC* gekennzeichnet ist und den Wert *C2* beinhaltet. Zuerst muss dem Monitor jedoch mitgeteilt werden, dass kein weiterer Befehl dem Programm hinzugefügt werden soll, denn im Moment erwartet der Monitor an Adresse *4003* einen weiteren Befehl. Um den Programmiermodus zu verlassen, muss einfach ***F*** eingegeben werden.

Nach der Bestätigung dieser Eingabe über die *RETURN*-Taste befinden wir uns wieder im Eingabemodus von SMON. Zusätzlich wird der eingegebene Code noch einmal aufgelistet und mit einer waagerechten Linie abgeschlossen.

Das Starten von Maschinenprogrammen

Jetzt wird es an der Zeit, das Programm zu starten und zu sehen, welche Auswirkungen es hat. Der Start erfolgt über die Eingabe von

G4000

Das ***G*** steht für *Go* und die nachfolgende Adresse die Startadresse des Maschinenprogramms. Nach der Eingabe und Bestätigung über die *RETURN*-Taste schaut das Ganze wie folgt aus.

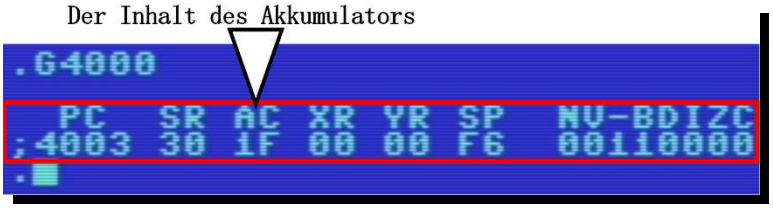
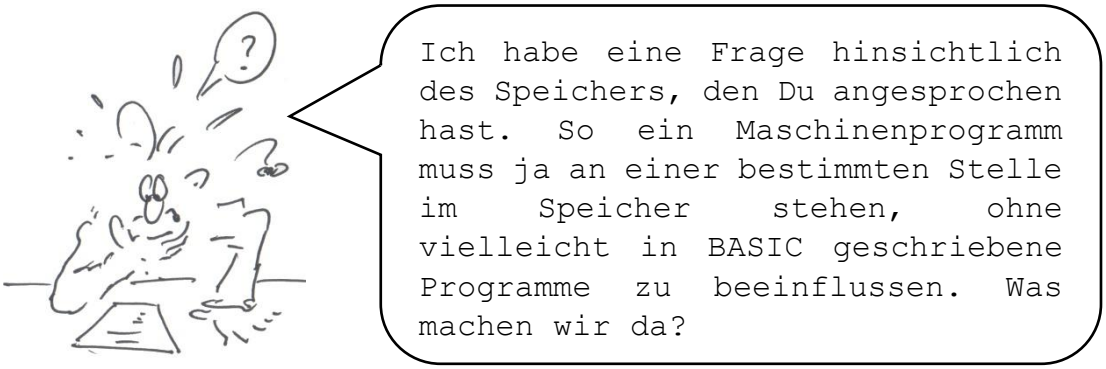


Abbildung 22 - Der Inhalt des Akkus lautet 1F

Und siehe da, der Inhalt hat sich auf *1F* geändert.



Vollkommen richtig erkannt! Ein Maschinenprogramm sollte nicht mit einem BASIC-Programm kollidieren, was natürlich höchstwahrscheinlich den Absturz des Computers zur Folge hätte. Das ist das Stichwort, auf das ich gewartet habe und eine gute Überleitung zum nächsten Kapitel. Wir schauen uns einmal die Speicheraufteilung des C64-Computers an.

Das Speichern und Laden von Programmen

Nun ist es sicherlich sinnvoll, einmal geschriebene Programm zu speichern. Sei es auf das Tapedeck oder auf Diskette. In einem späteren Kapitel „Wir zählen" zeige ich eine Variante, wo der Quellcode in einem Texteditor - zum Beispiel *Notepad++* - geschrieben wird, um dann den Code aus der Zwischenablage in VICE zu übernehmen. Eine feine Sache! Doch jetzt geht es darum, ein Programm aus SMON zu speichern und dann auch wieder zu laden. Ich nutze dafür das schon genutzte Diskettenlaufwerk, auf dem sich auch SMON befindet. Diese *ID* ist für dieses Floppy die *8* und standardmäßig so eingestellt. Für andere Device-Nummern muss über den ***I***-Befehl in SMON eine Anpassung vorgenommen werden. Also zum Beispiel über die Eingabe von

```
I 01
```

wird die Device-Nummer auf das Tape-Deck gelegt. Doch ich bleibe beim Diskettenlaufwerk *#8*. Angenommen, ich möchte das gerade geschriebene Programm

```
 4003 F
,4000  A9 1F     LDA #1F
,4002  00        BRK
--------------------------
.
```

speichern, so muss ich folgenden allgemeinen Befehl eingeben.

S"Name" ANFANGSADR ENDADR + 1

Für unseren konkreten Fall würde das wie folgt ausschauen. Die Startadresse ist natürlich *4000* und die Endadresse *4002 + 1*, also *4003*.

```
.S"PROGRAMM001" 4000 4003
```

Nach der Bestätigung über die *RETURN*-Taste erfolgt die folgende Anzeige.

```
.S"PROGRAMM001" 4000 4003
SAVING PROGRAMM001
.
```

Abbildung 23 – Das Programm wurde gespeichert

Das Programm wurde unter dem angegebenen Namen gespeichert. Wenn wir jetzt einmal kurz SMON über die Eingabe von ***X*** verlassen und das Inhaltsverzeichnis des Floppylaufwerks laden, dann sehen wir das zuvor gespeicherte Programm mit seinem Namen.

```
.X
READY.
LOAD"$",8

SEARCHING FOR $
LOADING
READY.
LIST

0 "                      " 00 2A
17    "SMONPC000",8,1      PRG
17    "SMONP3000",8,1      PRG
16    "SMONIC000",8,1      PRG
17    "SMONFC000",8,1      PRG
1     "ERIK001"            PRG
1     "ERIK002"            PRG
1     "PROGRAMM001"        PRG
594 BLOCKS FREE.
READY.
```

Abbildung 24 – Die Anzeige des Inhaltsverzeichnisses des Diskettenlaufwerks

Wir sehen als letzten Eintrag das zuvor gespeicherte Programm „PROGRAMM001". Zurück in SMON über die Eingabe von

SYS 49152

Um ein Programm von Diskette zurück in den Speicher zu laden, muss der allgemeine Befehl

L"Name"

oder

L"Name" STARTADR

eingegeben werden. Über die zweite Variante kann ein Programm an eine andere Startadresse im Speicher geladen werden, doch das ist hinsichtlich vorhandener Sprungadressen nicht unproblematisch und führt in der Regel zu Programmabstürzen, was bedeutet, dass der Rechner komplett eingefroren ist. Für das Laden meines Programms gebe ich also folgendes ein.

Nach der Bestätigung über die *RETURN*-Taste erfolgt die folgende Anzeige.

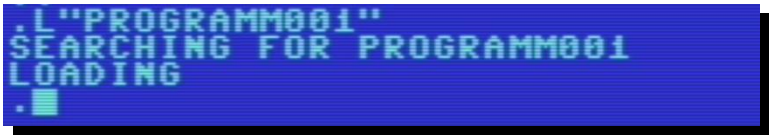


Abbildung 25 - Das Programm wurde geladen

Das Programm wurde unter dem angegebenen Namen geladen und kann ggf. bearbeitet oder einfach gestartet werden.

Wird versucht, ein Programm zu laden, das über den angegebenen Namen nicht auf dem Diskettenlaufwerk existiert, erfolgt keine Fehlermeldung in der Form „FILE NOT EXISTS". Es fehlt dann einfach die abschließende Meldung „LOADING".

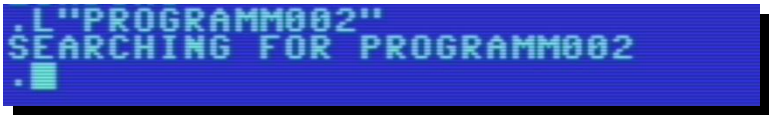


Abbildung 26 - Das Programm wurde unter dem angegebenen Namen nicht gefunden

Der Speicher des C64

Der Speicher des C64

Erinnerst Du Dich noch an den Adressbus des *6502*? Wie viele Adressleitungen stehen dem *6502* bzw. *6510* zur Verfügung? Es sind *16* Leitungen, was bedeutet, dass wir $2^{16} = 65.536$ unterschiedliche Adressen - also *64 KByte* - ansprechen können, in denen jeweils *8-Bit* breite Informationen abgelegt werden. In hexadezimaler Schreibweise bedeutet dies einen Adressraum von *$0000* bis *$FFFF*. Nun könnte man meinen, dass der komplette Speicherbereich zur freien Verfügung steht.

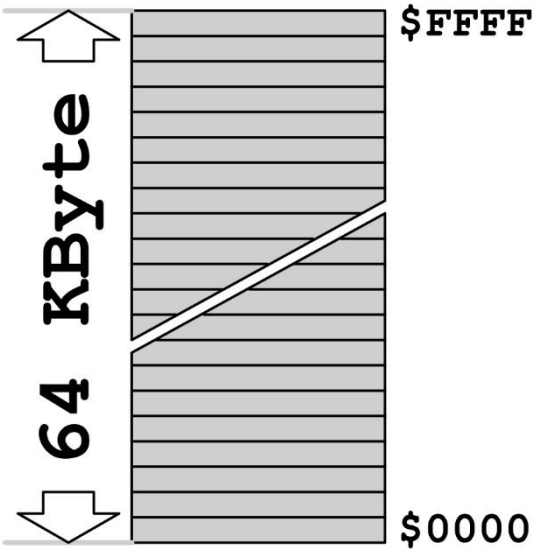


Abbildung 27 - Der komplette Adressraum einer 6502-CPU

Dem ist aber nicht so, denn an bestimmten Stellen stehen z.B. das Betriebssystem oder auch die Bereiche für vorgesehene Erweiterungen (ROM-Module). Das folgende Bild zeigt Dir die grundlegende Speicheraufteilung, die im Detail natürlich viel feiner ist, aber aus Platzgründen und wegen der fehlenden Übersichtlichkeit hier nicht gezeigt wird.

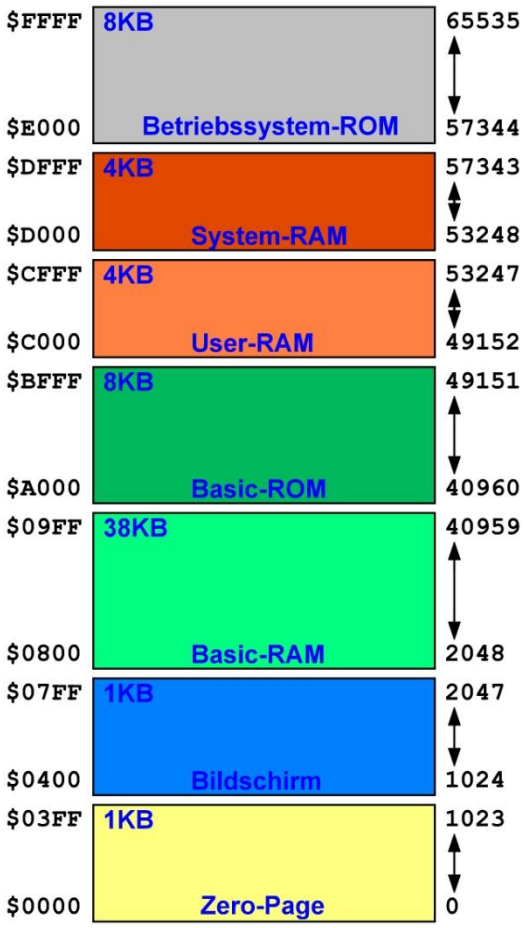


Abbildung 28 - Die grobe Speicheraufteilung des C64

Jeder einzelne Speicherblock ist farblich gekennzeichnet und mit Start- bzw. Ende-Adressen versehen. Auf der linken Seite befinden sich die Adressen im hexadezimalen, auf der rechten Seite im dezimalen Format. Lassen wir uns also ganz einfach beginnen und wahrlich keinen Stress aufkommen, denn es soll alles Spaß bereiten.

Der Aufruf von Assembler-Programmen

Der Aufruf von Assemblerprogrammen

Wenn man ein Assemblerprogramm aus dem Monitor heraus aufrufen möchte, dann haben wir gerade gesehen, dass das mit dem Befehl *G* für *Go* und der entsprechenden Adresse erfolgen muss. Also zum Beispiel *G4000*. Handelt es sich beim letzten Befehl um den *BRK*-Befehl, der eine Unterbrechung der Ausführung bewirkt, so verbleibt man im SMON-Monitor. Das folgende kurze Programm soll das noch einmal verdeutlichen.

Abbildung 29 - Der Aufruf eines Assemblerprogramms aus dem SMON-Monitor heraus

Nach der Abarbeitung ist zu sehen, dass sich der Cursor hinter dem *Punkt*-Prompt des Monitorprogramms befindet. Wird das Assemblerprogramm aus Basic heraus ausgerufen, was mit dem *SYS*-Befehl und der Angabe der Startadresse erfolgt, verbleibt man ebenfalls im Monitor. Der dezimale Wert *16384* steht hier für die Start- bzw. Einsprungadresse in das Assemblerprogramm, das ja ab *$4000* im Speicher zu finden ist. Der Programmzähler *PC* (***P**rogram-**C**ounter*) weist auf die nächste auszuführende Speicheradresse hinter dem *BRK*-Befehl.

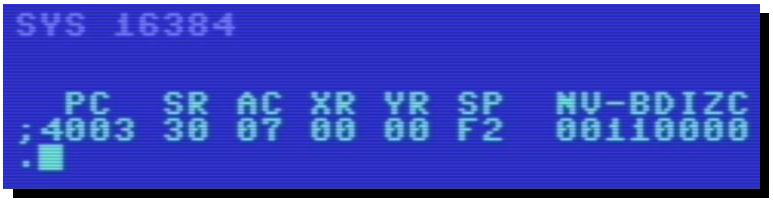


Abbildung 30 - Wir sind immer noch im SMON-Monitor

Ändern wir doch einmal den *BRK*-Befehl in einen *RTS*-Befehl ab. Dieser Befehl steht für ***R**e**T**urn from **S**ubroutine* und bedeutet übersetzt einen *Rücksprung aus dem Unterprogramm*. Das schaut dann wie folgt aus.

Abbildung 31 - RTS statt BRK

Rufen wir doch jetzt einmal das Assemblerprogramm aus BASIC heraus auf.

Abbildung 32 - Der Rücksprung zu Basic

Rechnen – Addieren und Subtrahieren

- Mit SMON rechnen
- Am Akkumulator führt kein Weg vorbei
- Die Addition
- Die Subtraktion
- Die Multiplikation
- Die Division

Rechnen - Addieren und Subtrahieren

Nun habe ich schon etwas über das Rechnen mit der CPU erzählt, doch das war eher ein Geplänkel am Rande, wie man das so schön nennt. Doch bevor es losgeht, möchte ich einige Stärken von SMON hinsichtlich Berechnungen und der Anzeige von Werten.

Mit SMON rechnen

Ich habe SMON gestartet und möchte ein paar Werte umrechnen. Also zum Beispiel Dezimalzahlen in Hexadezimalzahlen und umgekehrt. Gerade am Anfang ist die Handhabung und Umrechnung von Werten in unterschiedliche Zahlensysteme nicht so geläufig und bevor man sich einen Taschenrechner schnappt, um die Umrechnung zu starten, kann das auch mit Bordmitteln von SMON erfolgen.

Die Umrechnung von Hexadezimalzahlen bis $FF

Wie wir schon wissen, wird ein hexadezimaler Wert mit einem $-Zeichen eingeleitet. Geben wir doch einmal *$1F* ein und sehen, was das Ergebnis ist. Nachfolgend ist die Eingabe hinter dem Punkt noch nicht mit der *RETURN*-Taste bestätigt worden.

Abbildung 33 - Die Eingabe des hexadezimalen Wertes $1F

Nach der Bestätigung schaut das Ganze wie folgt aus. Es sind drei Werte zu sehen, wobei der erste der eingegebene *$1F* ist und mit *1F* dargestellt wird. Beim zweiten Wert handelt es sich um die Darstellung des Wertes in binärer Form *10110000* und der dritte Wert *31* ist die dezimale Umrechnung.

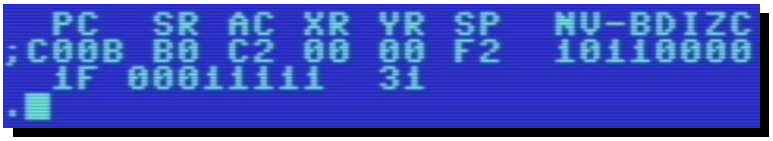


Abbildung 34 - Die Umrechnung des hexadezimalen Wertes $1F

Die Umrechnung von Dezimalzahlen bis 255

Natürlich geht das auch andersherum. Eine Dezimalzahl wird durch das #-Zeichen gekennzeichnet. Geben wir doch einmal *#134* ein und sehen, was das Ergebnis ist. Nachfolgend ist die Eingabe hinter dem Punkt noch nicht mit der *RETURN*-Taste bestätigt worden.

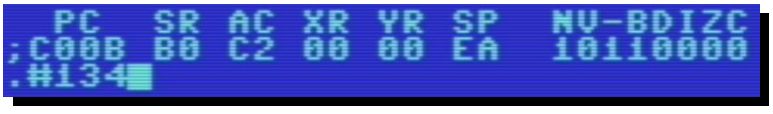


Abbildung 35 - Die Eingabe des dezimalen Wertes #134

Nach der Bestätigung schaut das Ganze wie folgt aus. Es sind wiederum drei Werte zu sehen, wobei der erste der umgerechnete

hexadezimale Wert ist und mit *86* dargestellt wird. Beim zweiten Wert handelt es sich um die Darstellung des Wertes in binärer Form *10000110* und beim dritten Wert *134* handelt es sich wieder um den eingegebenen und umzurechnenden Wert.

```
  PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 EA  10110000
  86 10000110  134
.
```

Abbildung 36 - Die Umrechnung des dezimalen Wertes #134

Die Umrechnung von Binärzahlen

Die Eingabe von Binärzahlen wird durch das %-Zeichen eingeleitet. Darüber können *8*-Bit Binärkombinationen eingegeben und umgerechnet werden. Geben wir doch einmal *%11000110* ein und sehen, was das Ergebnis ist. Nachfolgend ist die Eingabe hinter dem Punkt noch nicht mit der *RETURN*-Taste bestätigt worden.

```
  PC  SR AC XR YR SP  NV-BDIZC
;1359 30 00 00 00 E6  00110000
.%11000110
```

Abbildung 37 - Die Eingabe des binären Wertes %11000110

Nach der Bestätigung schaut das Ganze wie folgt aus. Es sind wiederum drei Werte zu sehen, wobei der erste der umgerechnete hexadezimale Wert ist und mit *C6* dargestellt wird. Beim zweiten Wert handelt es sich um die Darstellung des eingegebenen Wertes in binärer Form *11000110* und beim dritten Wert *198* handelt es sich wieder um den umgerechneten dezimalen Wert.

```
  PC  SR AC XR YR SP  NV-BDIZC
;1359 30 00 00 00 E6  00110000
  C6 11000110  198
.
```

Abbildung 38 - Die Umrechnung des binären Wertes %11000110

Etwas anders schaut es hinsichtlich umzurechnender Werte aus, die größer *255* sind. In diesem Fall erfolgt keine Anzeige der binären Konvertierung.

Die Umrechnung von Hexadezimalzahlen größer $FF

Geben wir zur Umrechnung eine hexadezimale Zahl größer *$FF* ein und sehen, was das Ergebnis ist. Nachfolgend ist die Eingabe hinter dem Punkt noch nicht mit der *RETURN*-Taste bestätigt worden.

```
  PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 DE  10110000
.$A000
```

Abbildung 39 - Die Eingabe des hexadezimalen Wertes größer $FF

Es ist nach der Bestätigung zu sehen, dass die Umrechnung in den entsprechenden dezimalen Wert erfolgt ist, jedoch die binäre Umwandlung fehlt.

```
   PC   SR AC XR YR SP   NV-BDIZC
;C00B  B0 C2 00 00 DE   10110000
A000   40960
.
```

Abbildung 40 - Die Umrechnung des hexadezimalen Wertes größer $FF

Ähnlich funktioniert das mit der Eingabe von dezimalen Werten größer *255*. Die Eingabe von entsprechenden Binärzahlen größer *11111111* funktioniert jedoch nicht.

Das Rechnen mit zwei hexadezimalen Zahlen

Über das Fragezeichen und der nachfolgenden Rechenoperation können zwei *4*-stellige hexadezimale Zahlen addiert oder subtrahiert werden. In dieser Weise ist eine Addition zu sehen.

```
.?000F+0001
  16 0001 0000   16
.
```

Abbildung 41 - Die Addition zweier 4-stelliger hexadezimaler Zahlen

Ist das Resultat kleiner *256*, wird zusätzlich in der mittleren Spalte der entsprechende Binärcode zur Anzeige gebracht. Nachfolgend ist eine Subtraktion zu sehen.

```
.?2000-0001
1FFF   8191
.
```

Abbildung 42 - Die Subtraktion zweier 4-stelliger hexadezimaler Zahlen

Da das Resultat größer *255* ist, werden nur der hexadezimale und der dezimale Wert angezeigt.

Am Akkumulator führt kein Weg vorbei

Es gibt hinsichtlich unterschiedlicher Operationen ein spezielles Register, das in den meisten CPUs eine entscheidende Rolle spielt. Es ist der Akku, den ich schon erwähnt hatte.

Dieser Akku dient als universelle Recheninstanz. Operationen, welche sich auf zwei Speicherstellen beziehen, müssen den Weg zuerst über den Akkumulator gehen. Somit ist der Akku eines der am meisten genutzten Register in der CPU. Doch was nützt es, wenn wir wissen, dass da ein Akku ist, der mit einzelnen Bits etwas anstellt und dann daraus irgendetwas macht, was als Ergebnis zur Verfügung steht? Man könnte das natürlich so stehen lassen, doch ich möchte darauf näher eingehen, denn das Rechnen mit Bits sollte verstanden werden. Natürlich könnte ich jetzt gewaltig ausholen und ein gewaltig dickes Buch mit *1000* Seiten Schreiben, wo es nur um Digitaltechnik und den entsprechenden

Grundlagen geht. Dazu gibt es genügend freies Material im Internet. Ich werde es aber recht kurz und knapp halten und mich auf Dinge und Bereiche konzentrieren, die für uns wichtig sind.

Die Grundlage beziehungsweise Basis ist natürlich das sogenannte Byte, was einen Zusammenschluss von acht einzelnen Bits bedeutet, wie das auf der folgenden Abbildung noch einmal zu sehen ist.

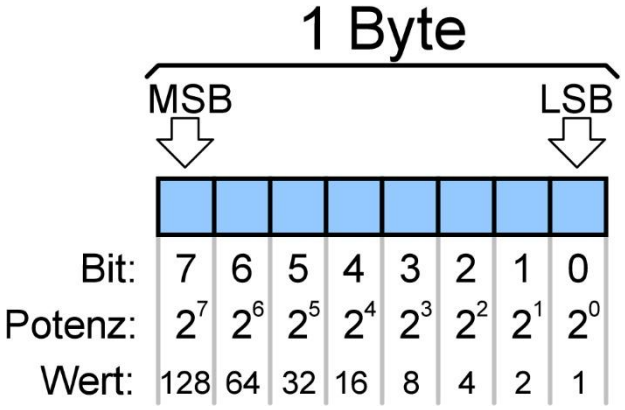


Abbildung 43 - Das Byte

Diese *8* Bits steigen in ihren Wertigkeiten von rechts nach links, was durch die gezeigte Potenzierung mit der Basis *2* je Stelle immer eine Verdopplung des Wertes bedeutet. Also *1*, *2*, *4*, *8*, usw. Zudem besitzen die ganz außen liegenden Bits, die quasi die Begrenzungen des Bytes repräsentieren, besondere Bezeichnungen. Ganz rechts außen ist es das niederwertigste Bit, was *LSB = Least Significant Bit* genannt wird und ganz links außen das höchstwertigste Bit mit der Bezeichnung *MSB = Most Significant Bit*. Die gleichen Bezeichnungen gibt es noch einmal im Verbund mit mehreren Bytes, wo es dann aber bei *LSB = Least Significant Byte* und *MSB = Most Significant Byte* lautet. Die folgenden Beispiele zeigen einige unterschiedliche Werte.

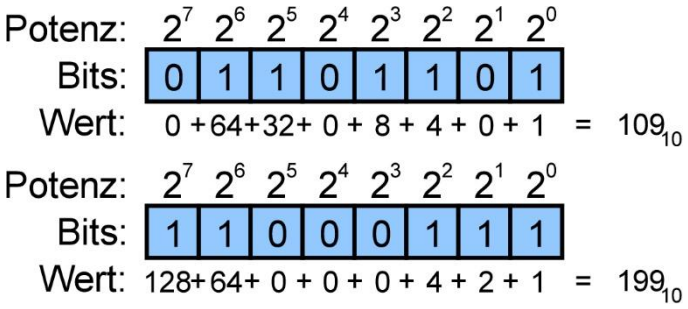


Abbildung 44 - Zwei Beispiele

Bevor ich jedoch beginne, sollte ich eine sehr einfache Vorgehensweise zeigen, mit der man Dezimalzahlen quasi zu Fuß in Binärzahlen wandeln kann. Ich höre schon die Kritiker schreien, dass es dafür doch Taschenrechner oder Programme wie *Calc* im *Programming-Mode* gibt. Schön!

Eine entsprechende Umrechnungsvorschrift entspricht der Division mit dem anfallenden Restwert. Da eine Zahl dividiert durch den Wert *2* immer nur den Restwert *0* oder *1* ergeben kann, entsteht daraus die äquivalente Binärzahl. Folgende Punkte müssen demnach abgearbeitet werden.

1. Die Zahl durch den Wert *2* dividieren
2. Den Restwert der Division notieren
3. Ist das Ganzzahl-Ergebnis nicht *0* ist, müssen die Schritte 1 und 2 wiederholt werden

Ich möchte die Dezimalzahl 150_{10} in eine Binärzahl wandeln.

Rechnung	Ganzzahl-Ergebnis	Restwert
150 ÷ 2 =	75	0
75 ÷ 2 =	37	1
37 ÷ 2 =	18	1
18 ÷ 2 =	9	0
9 ÷ 2 =	4	1
4 ÷ 2 =	2	0
2 ÷ 2 =	1	0
1 ÷ 2 =	0	1

Tabelle 5 - Die Umwandlung von Dezimal in Binär

Das Ergebnis erscheint nun in der Spalte Restwert, wobei das LSB ganz oben steht. Wir können also folgendes schreiben:

$150_{10} = 10010110_2$

Dieser kleine Einschub zur händischen Umrechnung von einer Dezimal- in eine Binärzahl sollte an dieser Stelle erst einmal genügen. Ich denke, ich sollte jetzt einfach einmal mit ein paar einfachen Rechenoperationen beginnen, damit klar ist, wie das mit den einzelnen Bits gehandhabt wird.

Positive und negative Werte

Wenn es darum geht, positive Werte zu speichern, dann sollten wir uns das auf binärer Ebene ansehen. Ich beziehe mit ich lediglich auf acht Bits. Das folgende Beispiel hatte ich gerade schon gezeigt.

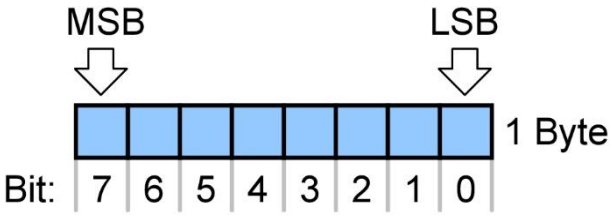


Abbildung 45 - Die acht Bits eines Bytes

Der Wertebereich erstreckt sich bekanntermaßen von *0* bis *255*. Um negative Werte zu kennzeichnen - ob man sie später also solche ansieht, kommt immer darauf an - wird auf Binärebene das

höchstwertige Bit auf *1* gesetzt. Diese Bit an Position 7 (Zählung beginnt bei *0*!) wird als sogenanntes *Vorzeichen-Bit* in Erscheinung treten. Es ist die Position, die auch *MSB*, also *Most Significant Bit*, genannt wird.

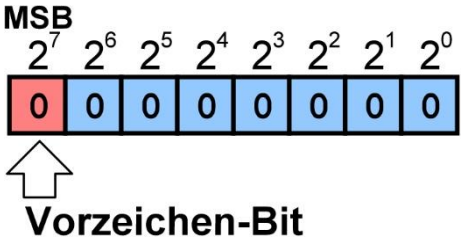


Abbildung 46 - Das MSB als Vorzeichen-Bit

Es liegt in der Natur der Sache, dass für die Abbildung des eigentlichen Wertes ein Bit weniger zur Speicherung zur Verfügung steht. Wie groß ist dann aber der zur Verfügung stehende Bereich, wenn zu den positiven Werten auch noch negative hinzukommen? Bei 8-Bits schaut das dann wie folgt aus.

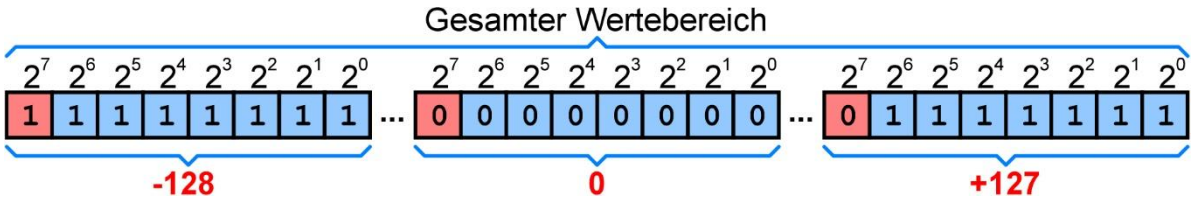


Abbildung 47 - Gesamter Wertebereich mit negativen Werten

Ich muss noch einmal betonen, dass es eine Interpretationssache ist, wie eine Bitkombination zu werden ist. Wird lediglich mit positiven Werten gearbeitet, dass ist der Wert *134* in der Bitkombination *10000110* vollkommen richtig. Das *Negative*-Flag wird dann einfach ignoriert. Doch wenn es darum geht, zusätzlich noch mit negativen Werten zu arbeiten, schaut die Welt vollkommen anders aus. Dann ist jedoch die gezeigte Bitkombination *10000110* vollkommen anders zu bewerten und entspricht nicht – wie man das vielleicht meinen möchte – dem Wert *-6*! Warum ist das so? Und warum entspricht aber die Bitkombination links außen *11111111* dem dezimalen Wert *-128* und wie kommt man dahin?

Es gibt zur Darstellung von negativen Werten das sogenannte *Einerkomplement* und das *Zweierkomplement*. Hört sich geschwollen an, ist aber recht einfach!

Was ist das Einerkomplement?

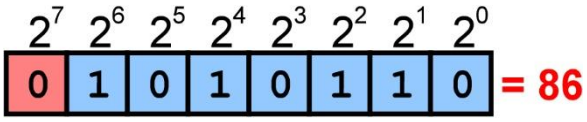
Das *Einerkomplement* ist eine arithmetische Operation im Binärsystem. Bei dieser Vorgehensweise werden alle Bits einer Binärzahl invertiert. Aus *0* wird *1* und aus *1* wird *0*.

Und dann...

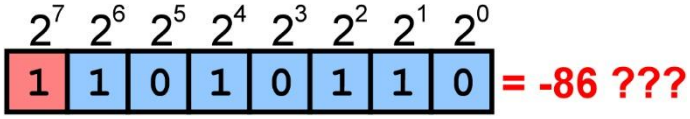
Was ist das Zweierkomplement?

Das *Zweierkomplement* stellt eine Möglichkeit dar, negative Zahlen im Binärsystem darzustellen, ohne die Angabe von zusätzlichen Zeichen wie + und – zu verwenden, da sich diese Information innerhalb der zur Verfügung stehenden Bitkombination verbirgt. Das Zweierkomplement wird aus dem Einerkomplement *+1* gebildet.

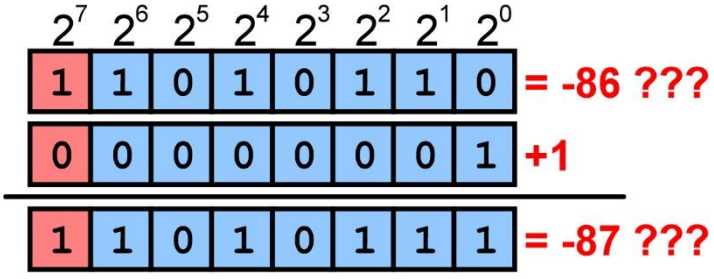
Warum muss das aber auf diese Weise geschehen? Dazu muss ich wieder ein wenig ausholen. Angenommen, wir haben die folgende Bitkombination vorliegen.



Diese positive Zahl (das MSB ist *0*) stellt einen dezimalen Wert von *+86* dar. OK, dann wollen wir daraus eine negative Zahl machen, indem wir lediglich das Vorzeichen-Bit von *0* auf *1* ändern. Das vermeintlich richtige Ergebnis wäre das folgende.



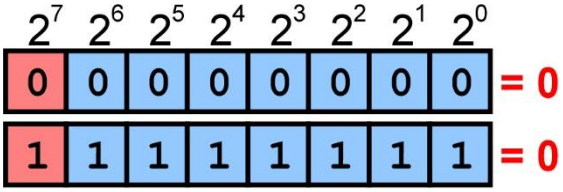
Wenn wir an dieser Stelle die Festlegung treffen würden, dass das der negative Wert der eben gezeigten positiven Zahl ist, wäre zunächst alles OK. Damit könnten wir leben. Doch in der Datenverarbeitung werden nicht nur Werte gespeichert und angezeigt. Es wird auch mit ihnen gerechnet. Und da laufen wir in ein Problem hinein. Wir nehmen einmal an, wir wollten einen Wert, sagen wir *+1*, addieren, was bedeutet, dass das Ergebnis um den Wert *1* größer wird als der Ursprungswert. Sehen wir uns das wieder auf Bit-Ebene an.



Na, fällt hier etwas auf? Trotz Addition eines positiven Wertes ist das Ergebnis um den Wert *1* kleiner geworden.

-86 + 1 = -87 ???

Auf diese Art und Weise kommen wir also nicht zum Ziel. Jetzt wenden wir das eben angepriesene Einerkomplement auf den Ursprungswert an. Ich werde dabei auch direkt auf das nächste Problem hinweisen, das sich bei einer ganz besonderen Zahl ergibt. Von jedem Wert kann ich das negative Pendant bilden, in dem ich ein negatives Vorzeichen davorsetze, so auch bei der Zahl *0*. Aber *0* und *-0* sind absolut identisch und es besteht kein arithmetischer Unterschied.

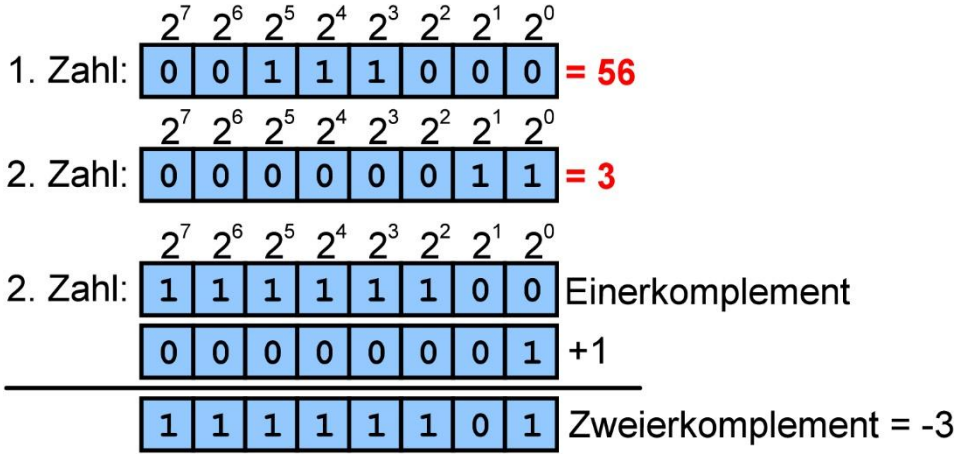


Das kann so nicht akzeptiert werden, da die Eindeutigkeit nicht gewährleistet ist. Aus diesem Grund wird der Wert *1* addiert, was in Summe die Zweierkomplementbildung ergibt.

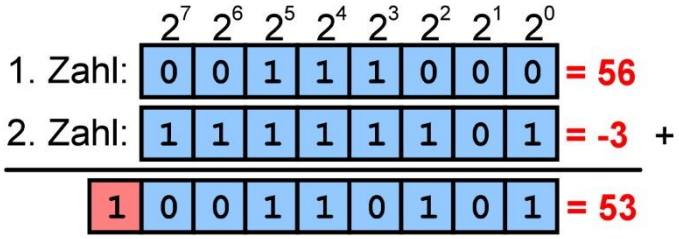
Dann wollen wir doch einmal eine einfache Subtraktion zuerst händisch und dann über die CPU durchführen, um zu sehen, ob da auch das gleiche Ergebnis herauskommt. Ich möchte die Differenz von *56* und *3* bilden. Wir kennen das folgende aus dem Schulunterricht.

Ergebnis = *Minuend* minus ***Subtrahend*** gleich Wert der **Differenz**

Um das händisch durchzuführen, kann man die Summe von beiden Werten bilden, doch zuvor muss vom Subtrahenden das Zweierkomplement gebildet werden. Das würde dann wie folgt ausschauen. Der Binärwert des Minuenden *56* steht ganz oben und das Zweierkomplement von *3* wird darunter gebildet.



Also muss nun lediglich die Summe von *00111000* (*56*) und *11111101* (*-3*) gebildet werden. Der entstandene Übertrag wird dabei nicht berücksichtigt!



Das Ergebnis lautet binär korrekterweise *00110101*, was dezimal *+53* entspricht. Abschließend zum Zweierkomplement zeige ich einmal den Zahlenkreis für *4*-Bit-Werte, in dem sowohl die positiven, als auch die negativen Werte aufgetragen sind.

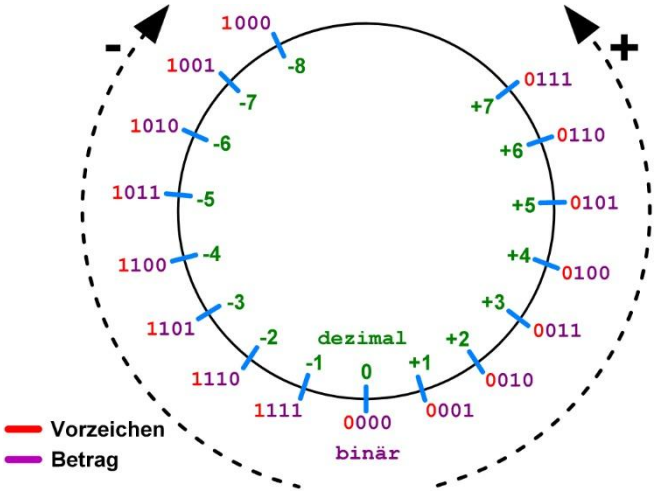


Abbildung 48 - Der Zahlenkreis für positive und negative Werte

Die Addition

Vielleicht erinnern sich einige noch an die Addition von Zahlen, wie wir sie in der Grundschule gelernt haben. Wir schreiben die Zahlen mit den einzelnen Stellen untereinander und addieren sie dann Stelle für Stelle. Dabei beginnen wir mit der letzten rechten Stelle und arbeiten uns dann langsam nach vorne durch. Die gleiche Vorgehensweise kommt zur Anwendung, wenn wir Binärzahlen – auch Dualzahlen genannt - addieren wollen. Ich werde das zu Beginn mal anhand von *4*-Bit-Zahlen erläutern. Die zugrunde liegenden Rechenregeln lauten wie folgt, wobei sich das natürlich nur auf eine einzige Stelle bezieht.

						Ü
	0	+	0	=	0	0
	0	+	1	=	1	0
	1	+	0	=	1	0
	1	+	1	=	0	1
1	+	1	+	1	=	1	1

Tabelle 6 - Die Rechenregeln für eine Addition

In einem einzelnen Additionsschritt werden immer nur zwei Werte addiert. Soll also eine Summe aus mehreren Werten gebildet

werden, dann addiert man zunächst die erste und den zweiten Wert, wobei das Ergebnis dann zum dritten Wert addiert wird. In dieser Weise wird fortgefahren, bis alle Werte addiert wurden. Eine Kolonnenaddition, wie man das von Dezimalzahlen gewohnt ist, ist bei Binärzahlen also nicht möglich, da es Probleme bei einem möglichen Übertrag gibt. Was ein Übertrag ist, wird gleich genauer beleuchtet, doch die meisten ahnen das schon, denn es bedeutet eine Berücksichtigung des Ergebnisses einer Stelle auf die nächste. Sehen wir uns also ein paar Beispiele zur Addition an. Nachfolgend werden die dezimalen Werte *10* und *1* addiert.

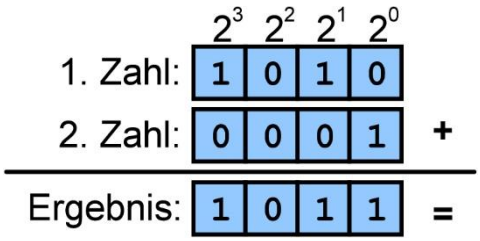


Abbildung 49 – Die Addition zweier Binärzahlen (ohne Übertrag)

Addiert wird natürlich von rechts nach links, wobei in diesem Beispiel noch kein Übertrag zustande kommt. In Prosa ausgedrückt lautet das dann.

- 0 + 1 gleich 1 und kein Übertrag auf die nächste Stelle
- 1 + 0 gleich 1 und kein Übertrag auf die nächste Stelle
- 0 + 0 gleich 0 und kein Übertrag auf die nächste Stelle
- 1 + 0 gleich 1 und kein Übertrag auf die nächste Stelle

Im nächsten Beispiel ist dann zu sehen, wie ein Übertrag zu handhaben ist. Es werden die dezimalen Werte *11* und *3* addiert.

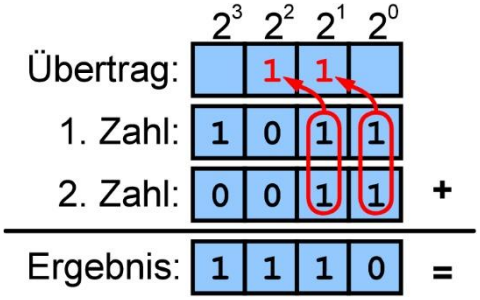


Abbildung 50 – Die Addition zweier Binärzahlen (mit Übertrag)

In Prosa ausgedrückt lautet das dann.

- 1 + 1 gleich 0 plus einen Übertrag auf die nächste Stelle
- 1 + 1 + 1 gleich 1 plus einen Übertrag auf die nächste Stelle
- 1 + 0 + 0 gleich 1 und kein Übertrag auf die nächste Stelle
- 1 + 0 gleich 1 und kein Übertrag auf die nächste Stelle

Mit diesem *4*-Bit-Beispielen kann man das Szenario auf jede mögliche Bit-Länge adaptieren. Ok verstehe, ich sollte doch mal was mit *8*-Bit-Werten machen. Ist auch eine gute Sache, denn die *6502*-CPU ist ja eine *8*-Bit-CPU und gerade, wenn es da zu einem Überlauf kommt, sieht es schlecht aus?! Nein, nicht wirklich! Aber sehen wir uns das an. Die beiden *8*-Bit-Werte sollen also addiert werden. Beim ersten Beispiel ist noch alles sozusagen im grünen Bereich, doch beim zweiten Beispiel gibt es das angesprochene Szenario, dass das Ergebnis nicht mehr in den Wertebereich von *8*-Bits hineinpasst. Wie groß ist dieser denn noch mal?

$$Anzahl\ Kombinationen = 2^{Anzahl\ Bits}$$

$$Anzahl\ Kombinationen = 2^8 = 256$$

Mit einer Datenbreite von *8* Bits können also *256* unterschiedliche Kombinationen erzielt werden. Das ist der Wertebereich von 0 bis *255*, nicht *256*! Beim folgenden Beispiel werden die Werte *198* und *58* einer Addition unterzogen.

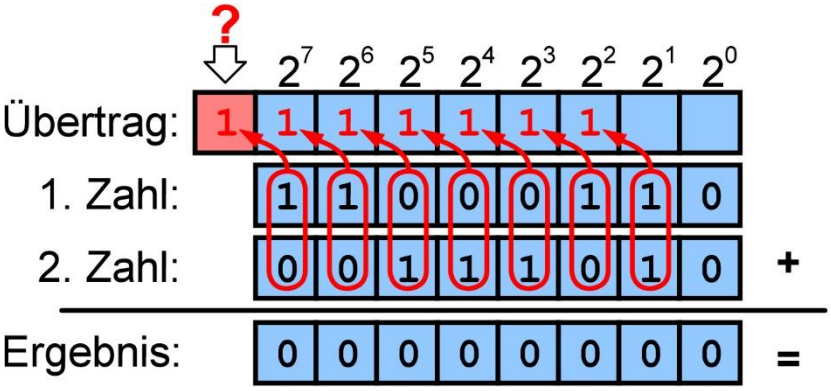


Abbildung 51 - Die Addition zweier Werte mit einem Überlauf

Natürlich habe ich die beiden Summanden nicht ohne Grund so gewählt, denn das Ergebnis der Addition von *198* und *58* ist genau *256*. Da mit einem einzigen Byte jedoch nur Werte bis maximal *255* abgebildet werden können, kommt es zwangsläufig zu einem Problem. Das Ergebnis ist 00000000_2, was einfach *0* bedeutet und nicht *256*. Die nächste binäre Stelle nach 2^7 wäre 2^8 und genau diese entspräche dem Wert *256*. Natürlich darf dieser Umstand eines Überlaufes auf keinen Fall unter den Tisch fallen und beim Design der *6502-/6510*-CPU wurde dieser Umstand natürlich berücksichtigt. Man hatte also zwei Möglichkeiten, diesem Dilemma zu begegnen.

- Den Datenbus breiter machen
- Ein Signal erstellen, dass den Überlauf kennzeichnet

Man hat sich für den zweiten Punkt entschieden. Dieses Signal ist im schon kurz erwähnten Status-Register als sogenanntes Flag (Flagge) zu finden, wie das auf der folgenden Abbildung zu sehen

ist, wobei in diesem Register noch weitere Flags vorhanden sind, die über andere Gegebenheit Aufschluss geben.

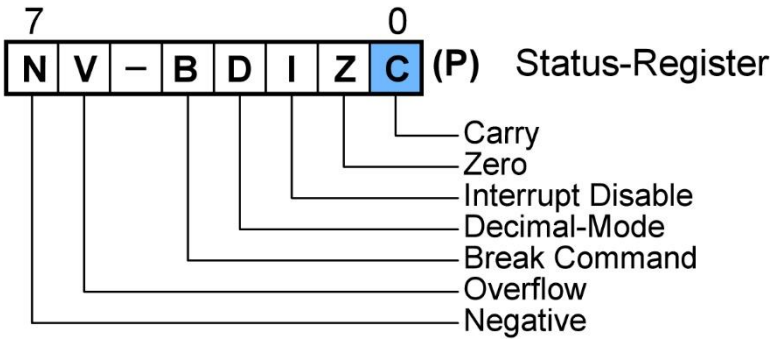


Abbildung 52 - Das Status-Register – Das Carry-Flag

Ich habe das für uns wichtige Bit ganz rechts außen farblich hervorgehoben. Dieses sogenannte Carry-Bit besitzt immer dann den Wert *1*, wenn es bei einer Rechenoperation zu einem Überlauf gekommen ist. Ich komme später noch dazu, wie das im Monitor des *von SMON* ausschaut. Die restlichen Bits des Status-Registers werden im Moment vernachlässigt und kommen später in eine nähere Betrachtung. Jetzt stellt sich erst einmal die Frage – und ich will ganz von vorne beginnen – mit welchen *OP-Code* eine Addition erzielt wird. Im nächsten Schritt sollte man sich dann Gedanken machen, wie denn dieses Carry-Bit angefragt oder sichtbar gemacht werden kann. Doch es ist erforderlich, noch einen Schritt zurück zu machen, denn bevor es an die angestrebte Rechenoperation geht, muss klar sein, wie die CPU das alles verarbeitet. Doch halt!

Die ALU und der Akku

ALU ist die Abkürzung für *Arithmetic Logic Unit* (Recheneinheit). Es handelt sich um eine Einheit in der CPU, die einen oder zwei numerische Werte annimmt, eine arithmetische Operation durchführt und dann das Ergebnis ausgibt. Die Operationen können von der einfachen Addition bis zu bitweisen Funktionen wie *OR*, *XOR* oder *AND* reichen. Er ist das Herzstück der Prozessorarchitektur. Moderne Prozessorarchitekturen können mehrere ALUs enthalten, wobei der *6502/6510* mit nur einer einzigen auskommt. Auf der folgenden Abbildung ist das Zusammenspiel zwischen ALU, Akku und dem Datenbus zu sehen.

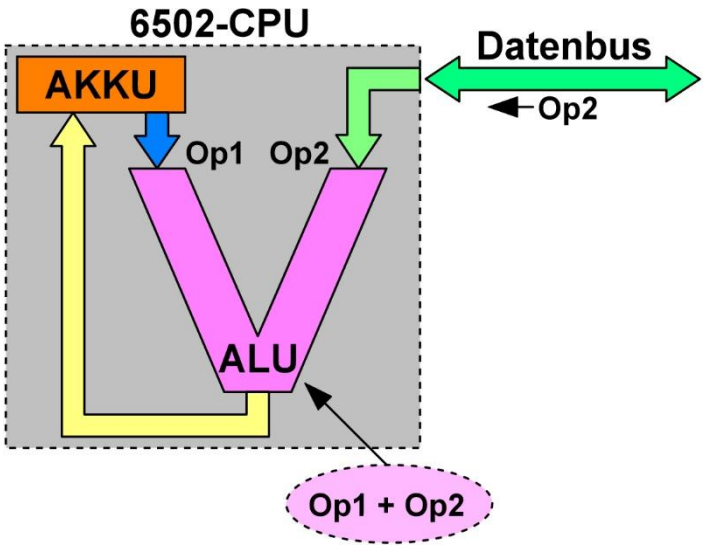


Abbildung 53 - Die Alu, der Akku und der Datenbus

Die Vorgehensweise bei einer Rechenoperation – hier eine Addition – ist wie folgt.

- Der Akku enthält den Operanden *Op1*
- Der Wert des Akkus gelangt an den linken Arm der ALU
- Der zweite Operand *Op2* gelangt vom Datenbus an den rechten Arm der ALU
- Die ALU führt die Addition durch
- Das Ergebnis der Rechenoperation wird zurück in den Akku transferiert

Hinsichtlich des Status-Registers kann gesagt werden, dass die Bits von der ALU in Abhängigkeit von der zuletzt durchgeführten Rechenoperation gesetzt werden. Gerade der schon erwähnte Überlauf wird bei Addition mit entsprechend großen Werten gesetzt.

Die verwendeten OP-Codes

So weit, so gut! Kommen wird nun zu den erforderlichen OP-Codes, um eine Addition durchzuführen. Einen Befehl haben wir natürlich schon kennengelernt. Es ist der Lade-Akku *LDA*-Befehl. Nun benötigen wir noch einen weiteren Befehl, der die Addition mit einem bestimmten Wert ermöglicht. Es gibt da mehrere Varianten, doch ich möchte das Beispiel mit einem festen Wert beginnen. Auf der folgenden Abbildung sind schon mal die entsprechenden Werte und das Ergebnis der angestrebten Addition zu sehen. Es werden die dezimalen Werte *9* und *3* addiert.

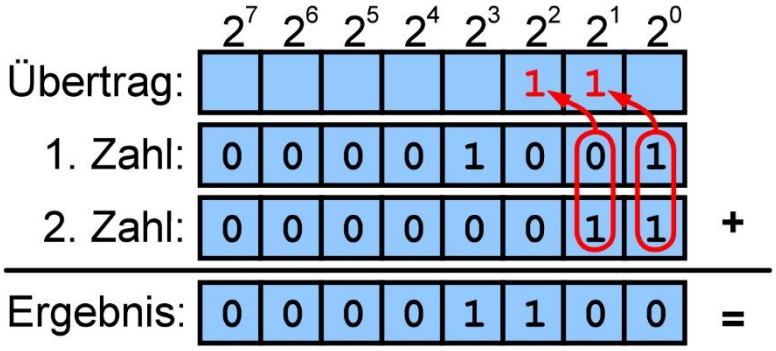


Abbildung 54 - Die Addition ohne Überlauf

Bevor ich zu den OP-Codes komme und wie sie überhaupt zustande kommen, zeige ich dir das Programm, das über den Monitor von *SMON* eingegeben wurde. Doch zuvor sollte ich noch kurz auf das Status-Register eingehen, wie es zur Anzeige gebracht wird.

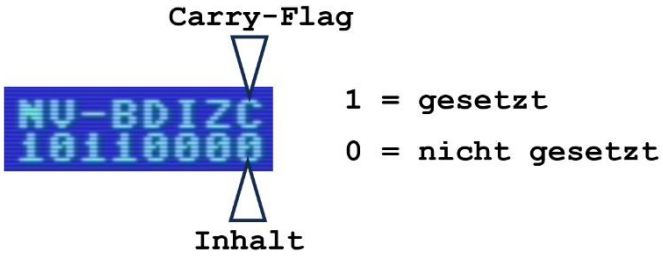


Abbildung 55 - Die Anzeige des Status-Registers

Es ist zu sehen, dass das Carry-Flag (C) nicht gesetzt ist. Es kann aber durchaus sein, dass dieses durch eine vorherige Operation gesetzt ist. Wird dieser Umstand vernachlässigt, kommt es bei nachfolgenden Rechenoperationen zu falschen Ergebnissen. Vor einer Rechenoperation sollte also auf jeden Fall sichergestellt sein, dass dieses Flag gelöscht ist. Das erfolgt über den *CLC*-Befehl, was *Clear Carry* bedeutet. Sehen wir uns das zuvor genauer an.

Das Carry-Flag löschen

Zu diesem Zweck habe ich das folgende kurze Programm geschrieben. Angenommen, das Carry-Flag ist gesetzt, wie das nachfolgend zu sehen ist.

Jetzt kann der erwähnte *CLC*-Befehl verwendet werden. Das Programm wird im Anschluss wieder mit dem *BRK*-Befehl unterbrochen.

```
  PC   SR AC XR YR SP   NV-BDIZC
;4002 31 1F 00 00 F6   00110001
.A4000
 4000  18          CLC
 4001  00          BRK
 4002 F
,4000  18          CLC
,4001  00          BRK
--------------------------------
.G4000

  PC   SR AC XR YR SP   NV-BDIZC
;4002 30 1F 00 00 F6   00110000
.
```

Abbildung 56 - Das Carry-Flag wird gelöscht

Nach dem Start des Programms mit

G4000

ist zu sehen, dass das Carry-Flag gelöscht wurde. Kommen wir jetzt zum eigentlichen Programm der Addition von zwei Werten.

Die Addition ohne Überlauf

Um die eigentliche Addition durchzuführen, wird der *ADC*-Befehl genutzt, der für *Add with carry* steht. Es wird also eine Addition unter der Berücksichtigung des Carry-Flags durchgeführt.

```
 4006 F
,4000  18          CLC
,4001  A9 09       LDA #09
,4003  69 03       ADC #03
,4005  00          BRK
--------------------------
.
```

Abbildung 57 - Das Programm zur Addition von 9 und 3

Das Programm wurde jedoch noch nicht ausgeführt. Starten wir das Programm mit

G4000

und sehen uns das Ergebnis an.

```
.G4000
  PC   SR AC XR YR SP   NV-BDIZC
;4006 30 0C 00 00 F6   00110000
.
```

Jetzt befindet sich der Wert *$0C* im Akku und wir realisieren, dass nach einer Rechenoperation sich das Ergebnis wieder im Akku befindet. Das Carry-Flag ist weiterhin nicht gesetzt, weil es zu keinem Überlauf gekommen ist.

Die OP-Codes

Werfen wir wieder einen Blick in den Monitor uns sehen uns diesen einmal genauer an.

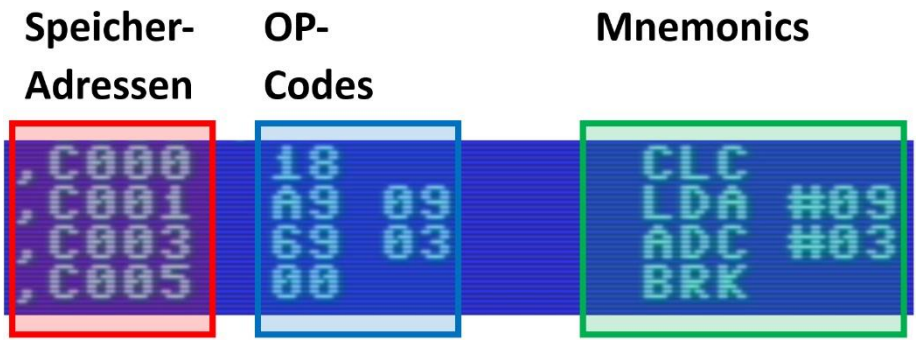


Abbildung 58 - Die unterschiedlichen Bereiche des Monitors

Die Anzeige ist in drei Spalten gegliedert

- Speicheradressen
- OP-Codes
- Mnemonics

In der mittleren Spalte sind die OP-Codes zu finden, die automatisch aus den eingegebenen Befehlen generiert werden. Hier ein paar Details dazu.

CLC:

Nachfolgend die Matrix für den *CLC*-Befehl.

OP-Code: 18

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2

Abbildung 59 - Die OP-Code-Matrixtabelle für den Op-Code CLC

Das ***i*** unterhalb des *CLC*-Befehls bedeutet, dass es sich um eine implizite Adressierung handelt.

N	V	-	B	D	I	Z	C
-	-	-	-	-	-	-	0

Tabelle 7 - Die Flag-Beeinflussung des CLC-Befehls

Operation: 0 → C (Wert 0 wird ins Carry übertragen)

Wir kommen später noch zu den unterschiedlichen Adressierungen der *6502*-CPU.

LDA:

Nachfolgend die Matrix für den *LDA*-Befehl durch eine Konstante, was durch den Lattenzaun (#) gekennzeichnet ist.

OP-Code: A9

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r •	6
7	BVS r	ADC (zp),y	ADC (zp) *		STZ zp,x •	ADC zp,x	ROR zp,x	RMB7 zp •	SEI i	ADC a,y	PLY s •		JMP (a,x) *	ADC a,x	ROR a,x	BBR7 r •	7
8	BRA r •	STA (zp,x)			STY zp	STA zp	STX zp	SMB0 zp •	DEY i	BIT #*	TXA i		STY a	STA a	STX a	BBS0 r •	8
9	BCC r	STA (zp),y	STA (zp) *		STY zp,x	STA zp,x	STX zp,y	SMB1 zp •	TYA i	STA a,y	TXS i		STZ a •	STA a,x	STZ a,x •	BBS1 r •	9
A	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp	SMB2 zp •	TAY i	LDA #	TAX i		LDY A	LDA a	LDX a	BBS2 r •	A

Abbildung 60 - Die OP-Code-Matrixtabelle für den Op-Code LDA

Das **#** unterhalb des *LDA*-Befehls bedeutet, dass es sich um eine unmittelbare (immediate) Adressierung handelt.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 8 - Die Flag-Beeinflussung des LDA-Befehls

Operation: M → A (Wert wird in den Akku gespeichert)

ADC:

Nachfolgend die Matrix für den *ADC*-Befehl durch eine Konstante, was durch den Lattenzaun (#) gekennzeichnet ist.

OP-Code: 69

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r •	6

Abbildung 61 - Die OP-Code-Matrixtabelle für den Op-Code ADC

Das **#** unterhalb des *ADC*-Befehls bedeutet, dass es sich um eine unmittelbare (immediate) Adressierung handelt.

N	V	-	B	D	I	Z	C
✓	✓	-	-	-	-	✓	✓

Tabelle 9 - Die Flag-Beeinflussung des ADC-Befehls

Operation: A + M + C → A, C (Addition von Akku + Wert + Carry. Das Ergebnis befindet sich wieder im Akku. Diese Operation beeinflusst das Carry-Flag)

BRK:

Nachfolgend die Matrix für den *BRK*-Befehl.

OP-Code: 00

MSD	W65C02S OpCode Matrix															MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0

Abbildung 62 - Die OP-Code-Matrixtabelle für den Op-Code BRK

Das ***s*** unterhalb des *BRK*-Befehls bedeutet, dass es sich um einen Befehl handelt, der u.a. auf den Stack wird. Es handelt sich dennoch um eine implizite Adressierung handelt.

N	V	-	B	D	I	Z	C
-	-	-	-	-	1	-	-

Tabelle 10 - Die Flag-Beeinflussung des BRK-Befehls

Wo kann man diese Informationen einsehen, die sich auf die OP-Code-Matrix beziehen?

	**Die OP-Code-Matrix des 6502**
https://www.westerndesigncenter.com/wdc/documentation/w65c02s.pdf	

Die Addition mit Überlauf

Wie schaut es jetzt bei einer Addition mit Überlauf aus? Sehen wir uns das wieder im Detail an. Es sollen die beiden dezimalen zahlen *198* und *61* addiert werden, was als Ergebnis *259* ist und nicht mehr in einem *8* Bit breiten Datenraum abgebildet werden kann.

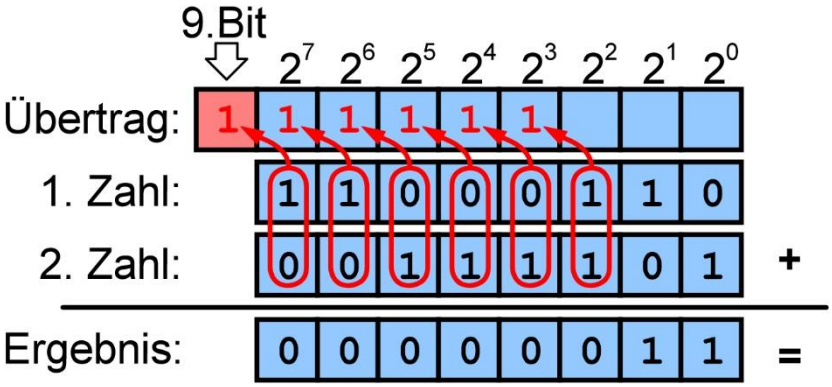


Abbildung 63 - Die Addition mit Überlauf

Wenn man sich das Ergebnis ohne den Überlauf anschaut, dann ist das der Wert *3*, was natürlich falsch ist. Das Carry-Flag muss mit einbezogen werden ist repräsentiert quasi ein *9tes* Bit. Das richtige Ergebnis lautet also.

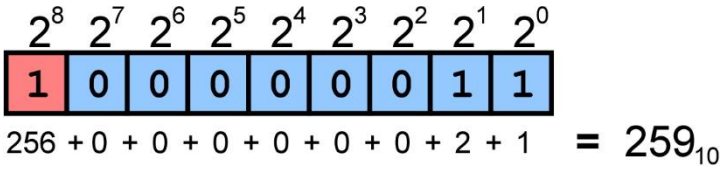


Abbildung 64 - Das richtige Ergebnis der Addition

Das entsprechende Assembler-Programm gestaltet sich wie folgt. Es ist zu sehen, dass das Carry-Flag nicht gesetzt ist. Ich habe das Programm wie gewohnt eingegeben und im Anschluss zur besseren Übersicht zuerst über die Eingabe von ***R*** die Registerinhalte anzeigen lassen und im Anschluss über ***D*** ein Disassemblieren mit der Angabe der Start- und Ende-Adresse vorgenommen.

```
.R
  PC   SR AC XR YR SP   NV-BDIZC
;4006 30 0C 00 00 F6   00110000
.D4000 4006
,4000   18          CLC
,4001   A9 C6       LDA #C6
,4003   69 3D       ADC #3D
,4005   00          BRK
--------------------------------
.
```

Nach der Ausführung schaut das anders aus. Das Carry-Flag wurde gesetzt und der Wert im Akku lautet wie erwartet *$03*, was als Ergebnis falsch ist.

```
.G4000
  PC   SR AC XR YR SP   NV-BDIZC
;4006 31 03 00 00 F6   00110001
.
```

Ein Tip für das Editieren in SMON

Hier ein wichtiger und sehr nützlicher Tip, wenn es um das Editieren von Code in SMON geht. Bei der Eingabe von Code steht am Anfang jeder Zeile ein unsichtbares Leerzeichen als Prompt. Wird diese Eingabe mit ***F*** verlassen und der Code später noch einmal angezeigt, befindet nun ein Komma am Anfang jeder Zeile, wie das auf der folgenden Abbildung zu sehen ist.

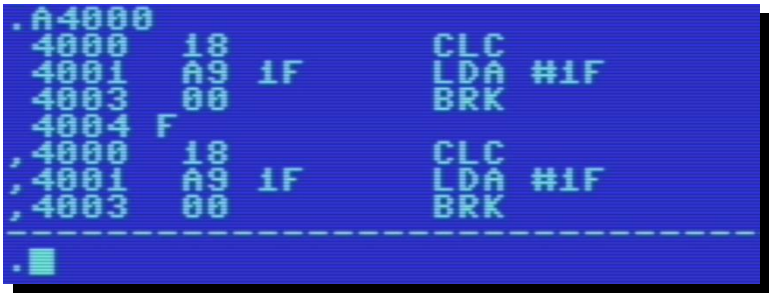


Abbildung 65 - Ein Tip für das Editieren in SMON

Nun kann mit den Cursor-Tasten in die Zeilen navigiert werden, um dort Änderungen an den Befehlen vorzunehmen. Nach der Bestätigung über die *RETURN*-Taste werden diese übernommen, so dass nicht alles noch einmal eingegeben werden muss. An der unmittelbaren Änderung des OP-Codes ist zu erkennen, dass diese Anpassungen unmittelbar Wirkungen zeigen. Der OP-Code kann auf diese Weise nicht unmittelbar geändert werden!

SMON verlassen

Um SMON zu verlassen und ins Basic zurückzukehren, muss lediglich ein ***X*** eingegeben und dieses mit der RETURN-Taste bestätigt werden.

Abbildung 66 - SMON wird beendet

Den SMON erneut aufrufen

Um SMON erneut aufzurufen, muss wieder der folgende Befehl aufgerufen werden.

SYS 49152

Abbildung 67 - SMON erneut starten

Unterschiedliche Adressierungsarten

Hier kommt nun erstmalig etwas ins Spiel, was die *6502-/6510-*CPU sehr stark macht. Es handelt sich um verschiedene Adressierungsarten. Einen Schimmer, was das sein könnte? Wenn wir zum Beispiel den Befehl zum Laden eines Wertes in den Akku verwendet haben, dann ist da der *Befehl* und der sogenannte *Operand*.

Abbildung 68 - Befehl und Operand

Die Adressierung bezieht sich nun auf die Angabe im besagten Befehl, wo der Operand überhaupt zu finden ist, mit dem dann eine Operation durchgeführt werden soll. Ich möchte hier einleitend einmal zwei unterschiedliche Adressierungsarten vorstellen, wobei die erste schon – ohne sie zu benennen – mehrfach zum Einsatz gekommen ist. Es handelt sich dabei um die sogenannte *unmittelbare Adressierung*, die *Immediate* genannt wird.

Die unmittelbare Adressierung - Immediate

Fangen wir mit der unmittelbaren Adressierung an.

	Die unmittelbare Adressierung (immediate)
Bei der unmittelbaren Adressierung steht hinter dem eigentlichen Befehl eine Konstante, die mit # gekennzeichnet ist.	

Dem *8*-Bit langen Befehlscode wird dabei eine acht Bit lange Konstante – auch *Literal* genannt – angehängt. Verschiedene Beispiele zur unmittelbaren Adressierung schauen wie folgt aus.

```
.A4000
 4000  A9 15     LDA #15
 4002  A9 2B     LDA #2B
 4004  A2 15     LDX #15
 4006  A2 2B     LDX #2B
 4008
```

Abbildung 69 - Unmittelbare Adressierung bei LDA und LDX

Die Schreibeweise mit dem anführenden Hash # (auch Lattenzaun genannt) symbolisiert die unmittelbare Adressierung. Es ist hier gut zu erkennen, dass sich der OP-Code zwischen *LDA* und *LDX* unterscheidet, denn der Prozessor muss ja wissen, in welches

Register der Wert gespeichert werden soll. Sehen wir kurz in der OP-Code-Matrix nach.

LDA (#) unmittelbar - A9 **LDX (#) unmittelbar - A2**

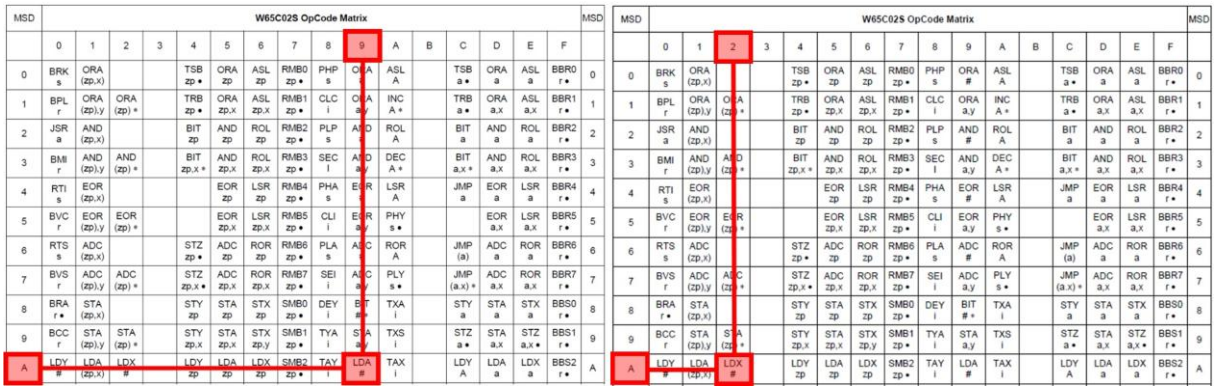


Abbildung 70 - Die OP-Codes für LDA und LDX für die unmittelbare Adressierung

Auch der gerade hinzugekommene Befehl *ADC* nutzt diese Art der Adressierung. Also zum Beispiel.

```
4008   69 17     ADC #17
400A   69 7C     ADC #7C
400C
```

Abbildung 71 - Unmittelbare Adressierung bei ADC

Sehen wir uns auch hier noch einmal den entsprechenden OP-Code zu *ADC* für die unmittelbare Adressierung an, der *$69* lautet.

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r •	6
7	BVS r	ADC (zp),y	ADC (zp) *		STZ zp,x •	ADC zp,x	ROR zp,x	RMB7 zp •	SEI i	ADC a,y	PLY s •		JMP (a,x) *	ADC a,x	ROR a,x	BBR7 r •	7
8	BRA r •	STA (zp,x)			STY zp	STA zp	STX zp	SMB0 zp •	DEY i	BIT # *	TXA i		STY a	STA a	STX a	BBS0 r •	8
9	BCC r	STA (zp),y	STA (zp) *		STY zp,x	STA zp,x	STX zp,y	SMB1 zp •	TYA i	STA a,y	TXS i		STZ a •	STA a,x	STZ a,x •	BBS1 r •	9
A	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp	SMB2 zp •	TAY i	LDA #	TAX i		LDY A	LDA a	LDX a	BBS2 r •	A

Abbildung 72 - Der OP-Code für ADC für die unmittelbare Adressierung

Was kennzeichnet die unmittelbare Adressierung?

Bei der unmittelbaren Adressierung werden die Daten – auch Operand genannt – die dem OP-Code folgen, unmittelbar in das betreffende Register geladen. Beispiele hierfür sind Akku, X-Register oder Y-Register. Das #-Zeichen deutet auf eine unmittelbare Adressierung hin.

Schauen wir uns nun eine weitere Möglichkeit der Adressierung an.

Die absolute Adressierung - Absolute

Nun ist es nicht immer sinnvoll, mit festen Werten in Form von Literalen zu arbeiten. Gewisse Prozesse oder Algorithmen führen Berechnungen durch, die dann ihre Ergebnisse irgendwo im Speicher ablegen. Es muss dann auch möglich sein, diese Werte wieder aus dem Speicher zu lesen, um damit zu arbeiten. Auf der nachfolgenden Abbildung ist ein Lesevorgang über die absolute Adressierung zu sehen.

Die absolute Adressierung

Bei der unmittelbaren Adressierung steht hinter dem eigentlichen Befehl eine Adresse für eine Position im Speicher. Für eine *16*-Bit-Adresse ist es wichtig zu wissen, dass die Darstellung im Low-/High-Byte-Format erfolgt. Hier werden *3* Bytes bestehend aus einem Byte OP-Code und zwei Bytes für die Operanden benötigt. Ein Befehl wie ***LDA $0400*** ist im Speicher in der folgenden Form zu finden: ***AD 00 04***.

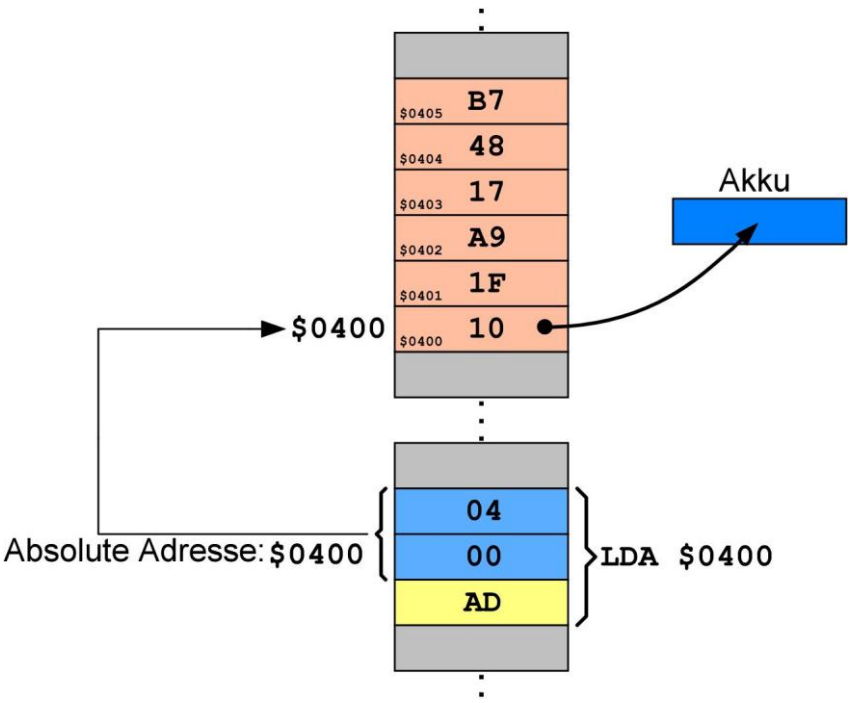


Abbildung 73 - Die Absolute Adressierung

Über den Befehl

LDA $0400

wird nun nicht ein fester Wert in den Akku geladen, sondern der Wert, der sich in der angegebenen Adresse befindet. Und diese Adresse lautet in dem Fall *$0400*. Es handelt sich in diesem Fall um einen *3* Byte-Befehl. Diese Adresse, die in dezimaler Schreibweise *1024* lautet, ist eine besondere Adresse beim C64. Es handelt sich dabei um die erste Adresse des Bildschirmspeichers in der linken oberen Ecke. Das Ganze schaut wie folgt aus.

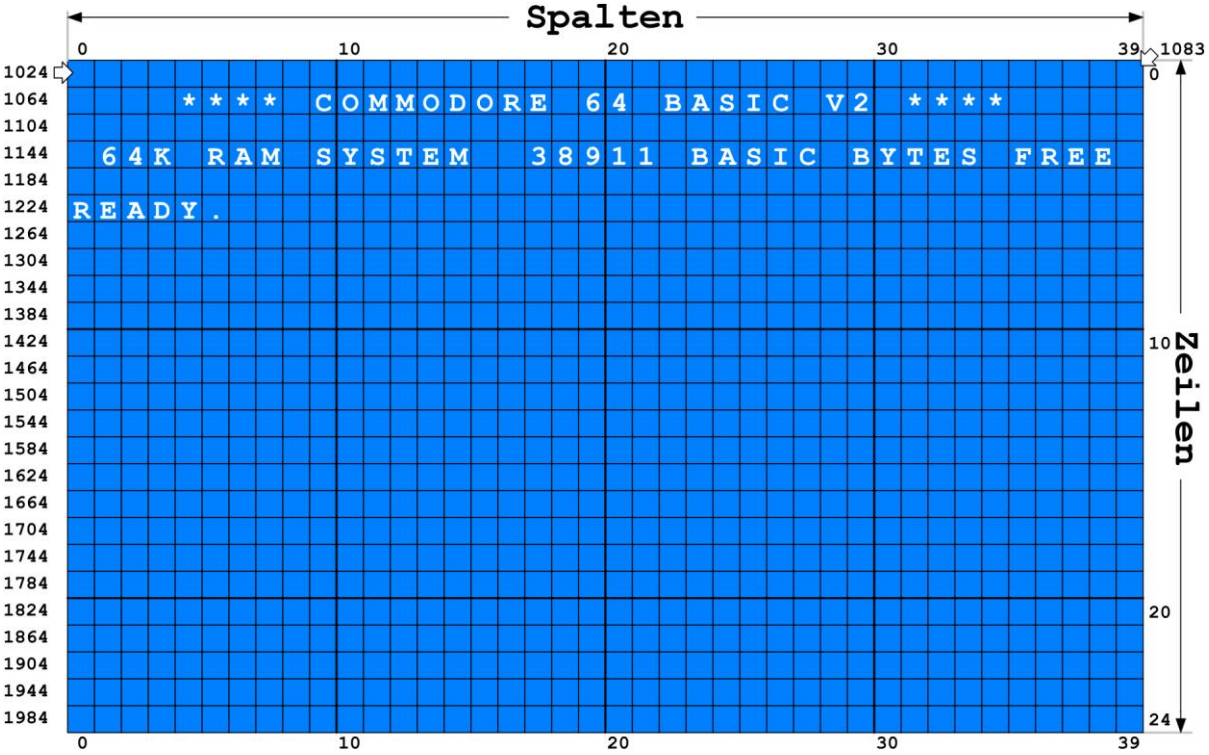


Abbildung 74 - Der Bildschirmspeicher des C64

Es sind also *40x25=1000* Zeichen verfügbar. Für den folgenden Test ist es wichtig zu wissen, wie der komplette Bildschirm gelöscht werden kann. Dazu muss aus Basic heraus das folgende *Print*-Kommando abgesetzt werden.

```
? CHR$(147)
```

Nach der Ausführung ist der Bildschirm nahezu leer und der Cursor befindet sich an erster Position in Spalte *1* bzw. in Zeile *4*. Die Zählung beginnt zwar normalerweise mit dem Index *0*, doch das ist hier erst einmal zu vernachlässigen.

Abbildung 75 - Der Cursor nach einem Löschen des Bildschirms

Um einen komplett leeren Bildschirm zu erhalten, muss auf einem C64 die Tastenkombination ***SHIFT + CLR/HOME*** gedrückt werden. In VICE ist das die Tastenkombination ***SHIFT + POS1***.

Kommen wir nun zum eigentlichen Thema zurück. Wir wollen mit einem *LDA*-Befehl die linke obere Position des Bildschirmspeichers auslesen und in den Akku schreiben. Ich habe also mit der Tastenkombination ***SHIFT + POS1*** den Bildschirm komplett geleert und SMON mit dem *SYS*-Befehl gestartet.

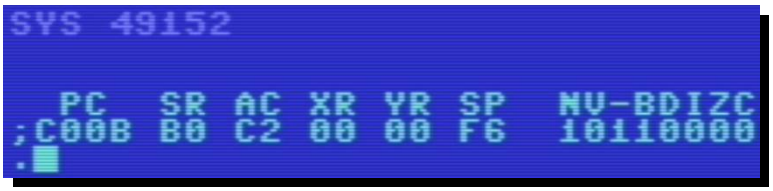


Abbildung 76 - SMON wird gestartet

Im nächsten Schritt geben wir das folgende Programm ein.

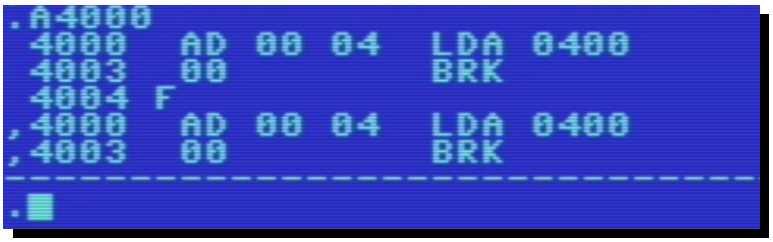


Abbildung 77 - Die Abfrage der ersten Bildschirmspeicherstelle ($0400)

Es sei zu bemerken, dass bei der Eingabe die folgende Schreibweise genutzt wurde.

LDA $0400

SMON hat dies, wie zu sehen ist, umgewandelt. Würde es sich bei einem Wert um eine Konstante handelt, sähen wir das vorangestellte #-Zeichen. Ok, dann starte ich jetzt einmal das Programm über die Eingabe von

G4000

und wir sehen, welchen Inhalt des Akkus zeigt.

Abbildung 78 - Das Programm wird gestartet

Der Inhalt von *AC* ist jetzt 13. An der ersten Bildschirmposition befindet sich das ***S*** und es handelt sich um den *19*. Buchstaben im Alphabet. Wie passt das mit der *19* überein? Ganz einfach! Wir erinnern uns, dass im SMON nur hexadezimale Werte dargestellt werden und der Wert *13* ist hexadezimal, was der dezimalen Zahl *19* entspricht. Wir können jetzt einige Versuche mit anderen Buchstaben an der ersten Bildschirmposition machen. Einfach mit dem Cursor hoch navigieren und z.B. ein ***A*** eintippen. Dann wieder in die Zeile mit dem Befehl

G4000

herunterwandern und die *RETURN*-Taste drücken.

Abbildung 79 - Das Programm wurde erneut ausgeführt

Nun ist in *AC* der Wert *01* zu sehen, was dem Code des Buchstabens ***A*** entspricht. Etwas scheint aber für das ungeübte Auge vielleicht merkwürdig erscheinen. Es wurde die Speicheradresse

$0400

eingegeben, doch in der Anzeige ist folgendes zu sehen.

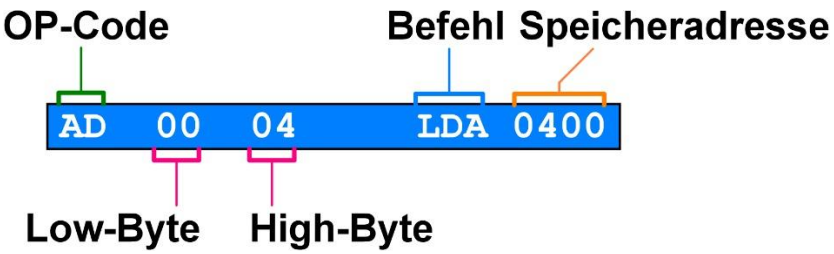


Abbildung 80 - Low- und High-Byte

Die Anzeige hinsichtlich der Adresse beträgt ***00 04***. Das hängt mit folgendem zusammen. In 8-Bit-Computersystemen wie dem *C64* hat der Adressbus eine Adressbreite von *16* Bit, um so *64* KByte Speicher adressieren zu können. Es werden also zwei *2* Bytes benötigt, um eine vollständige *16*-Bit-Speicheradresse anzusprechen. Die Standardreihenfolge für Speicheradressbytes ist das niederwertige Byte (Low-Byte - auch LSB) zuerst, gefolgt von dem höherwertigen Byte (High-Byte - auch MSB). So wird beispielsweise die absolute Speicheradresse *$A000* in zwei aufeinanderfolgenden RAM-Bytes als *$00*, gefolgt von *$A0* gespeichert.

Die implizite Adressierung

Bei der impliziten Adressierung scheint der Zusammenhang zuerst nicht ganz klar, denn eigentlich wird hier nichts adressiert.

Die implizite Adressierung

Die implizite Adressierung besteht lediglich aus einem einzigen Byte. Im Grunde genommen handelt es sich nur um den OP-Code ohne einen Operanden. Das sind Befehle wie *BRK*, *INX*, *DEY*, usw.

Die Zero-Page Adressierung

Bei dieser Adressierungsart muss ich zuvor ein wenig ausholen, denn es sollte ein Verständnis darüber bestehen, wie bestimmte Bereich oder Teile des Speichers aufgeteilt sind. Im Grunde ist es aber recht einfach.

Die Zero-Page Adressierung

Die Zero Page Adressierung handelt gleicht der absoluten Adressierung. Hier werden jedoch nur zwei Bytes bestehend aus einem Byte Opcode und einem Byte Operand benötigt.

Bei der *6502-/6510*-CPU wird der Adressraum des *16*-Bit breiten Adressbusses in logische Seiten (Pages) unterteilt. Logisch bedeutet also eine – man kann sagen – willkürliche Unterteilung, die auch anders hätte sein können und mit der realen Hardware eigentlich nicht direkt etwas zu tun hat. Eine einzige Speicherseite stellt dabei einen zusammenhängenden Block von 256 Speicherstellen dar, wie das auf der folgenden Abbildung zu erkennen ist.

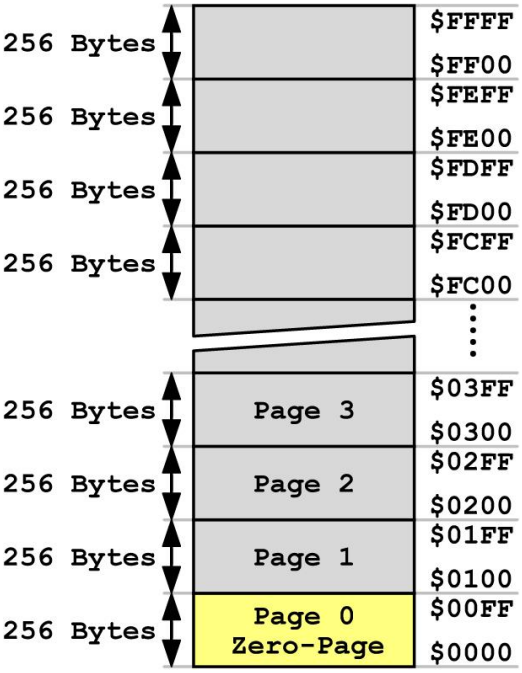


Abbildung 81 – Die Unterteilung des Adressraums in logische Seiten

Ein besonderes Augenmerk liegt hierbei auf Seite *0*, die *Zero-Page* genannt wird und sich von *$00* bis *$FF* erstreckt. Diese Seite besitzt eine besondere Bedeutung, denn sie wird bei der sogenannten Zero-Page-Adressierung verwendet. Diese Adressierung ist eine Sonderform der absoluten Adressierung. Sehen wir uns diese beiden Adressierungsarten in der Gegenüberstellung an.

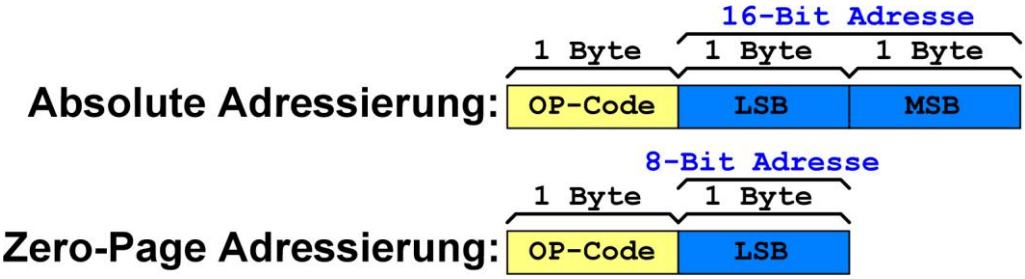


Abbildung 82 – Die Gegenüberstellung von absoluter und Zero-Page-Adressierung

Im Gegensatz zur 16-Bit Adresse bei der absoluten Adressierung nutzt die Zero-Page-Adressierung quasi eine Kurzadresse von nur 8-Bits und bedeutet eine Geschwindigkeitssteigerung, da weniger Programmcode respektive Bytes benötigt werden. Wie wir ja schon

kennengelernt haben, besteht eine Adresse normalerweise aus der Zusammensetzung von zwei Bytes, dem *LSB* und *MSB*. Für den Adressbereich von *0* bis *255*, also *$0* bis *$FF*, ist das aber nicht erforderlich, da ein einziges Byte dafür ausreicht, weil das MSB in diesem Fall immer *0* wäre und aus diesem Grund einfach weggelassen wird. Auf der nachfolgenden Abbildung ist ein Lesevorgang über die absolute Zero-Page-Adressierung zu sehen.

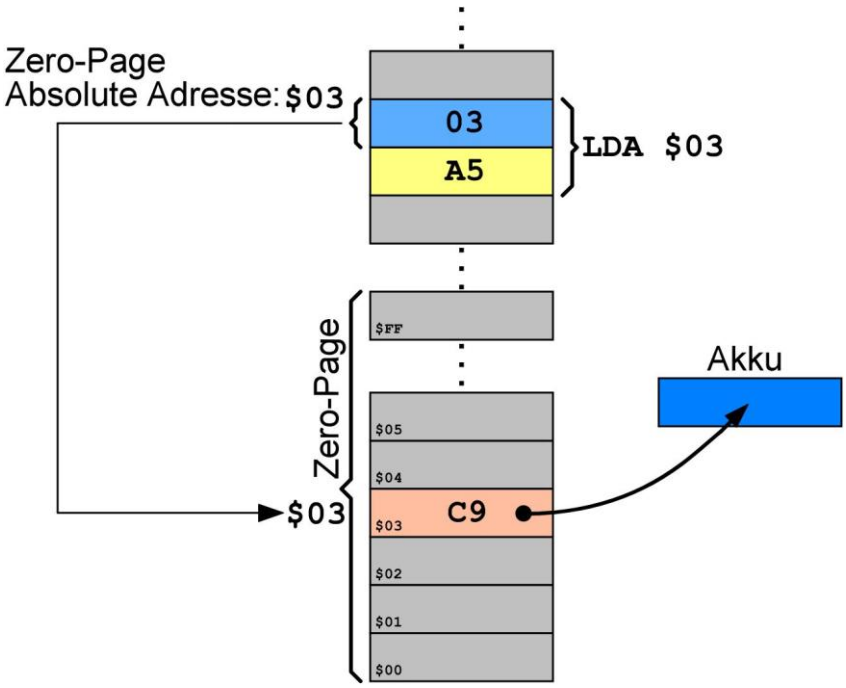


Abbildung 83 - Die Absolute Zero-Page-Adressierung

Die Indizierte Adressierung - Teil 1

OP-Codes mit einer indizierten Adressierung gestatten es, eine Folge von Speicheradressen für den Zugriff auf aufeinanderfolgende Speicherstellen zu bilden. Wie der Ausdruck *Indiziert* beziehungsweise Index schon vermuten lässt, handelt es sich dabei um eine Möglichkeit, auf verschiedene Speicher zu Zeigen.

Die indizierte Adressierung

Das Prinzip indizierter Adressierung besteht darin, dass ein Befehl sowohl eine Adresse als auch ein Indexregister angibt. Im allgemeinen Fall wird der Inhalt des Indexregisters zur Adresse addiert was so die Endadresse ergibt.

Wenn man Zeigefinger ins Englische übersetzt, lautet das *Index-Finger*. Hinsichtlich der indizierten Adressierung bedeutet das konkret, dass zu einer festgelegten Anfangsadresse ein sich stetig steigender (Inkrementierung) oder fallender

(Dekrementierung) Wert hinzuaddiert wird. Dafür eignen sich wunderbar das *X*- oder das *Y*-Register, die aus diesem Grund auch Index-Register genannt werden. Für die indizierte Adressierung gibt es verschieden Möglichkeiten.

- Absolut-Indizierte-Adressierung
- Zero-Page-Indizierung
- Indiziert-Indirekte-Adressierung
- Indirekt-Indizierte-Adressierung

In dieser Adressierungsart enthält der OP-Code eine absolute Adresse, die im Zusammenspiel mit dem *X*- oder *Y*-Register die tatsächliche Adresse des gewünschten Speicherplatzes indiziert. Als Beispiel für das *X*-Register zeige ich den *INX*-Befehl, der den Inhalt des *X*-Registers inkrementiert. Als Anfangsadresse ist jede Speicheradresse von *$0000* bis *$FFFF* möglich. Die Überschrift mit dem Zusatz - *Teil 1* dieses Kapitel lässt vermuten, dass es auch einen zweiten Teil gibt, den ich dann zur Einleitung einer weiteren Adressierungsart, die *indirekt-indizierte-Adressierung* nutzen möchte. Wenn es zum Beispiel um das Ansprechen der Bildschirmpositionen mit seinen 1000 Speicherstellen geht, so ist das mit der indizierten Adressierung und seiner Schrittweite von *256* Bytes (*0* bis *255*) nicht so einfach. Es können nicht alle Bildschirmadressen ohne eine neue Startadresse erreicht werden. Dazu später mehr. Kommen wir nun zurück zur indizierten Adressierung.

Hier ein einfaches Beispiel. Mit dem Befehl

LDA oper,x

werden die Inhalte der Speicheradressen oper+x in den Akku geladen. Vor jeder erneuten Ausführung von *LDA oper,x* sollte der Inhalt des *X*-Registers zum Beispiel inkrementiert werden, so dass daraus eine Folge von Adressen gebildet wird, die es anzusprechen gilt.

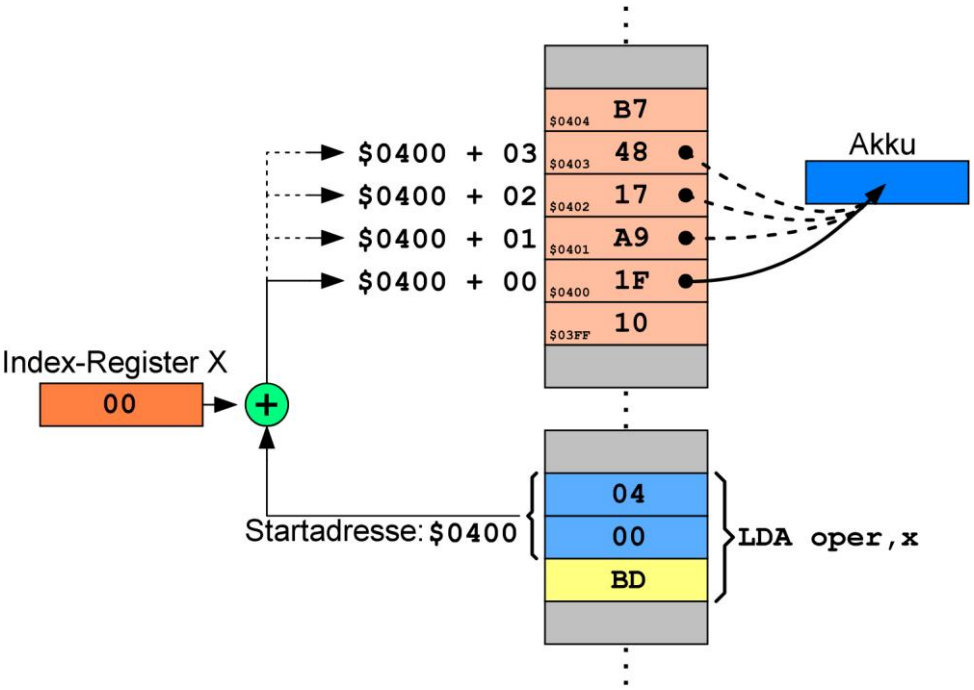


Abbildung 84 - Die absolute Indizierte Adressierung

Nun können wir das auch mal an einem konkreten Beispiel testen, denn über diesen Ansatz ist es möglich, bestimmte Speicherstellen hintereinander abzufragen und zu verarbeiten. natürlich nehmen wir hier wieder unseren schon verwendeten Bildschirmspeicher, der ja ab Adresse *$0400* beginnt. Bei diesem Beispiel kommen zudem neue Befehle hinzu und das wird spannend! Ziel ist es, die ersten *40* Bildschirmadressen abzufragen und deren Inhalte an neue Positionen hinzukopieren. Das hört sich vielleicht kompliziert an, doch wenn wir schrittweise vorgehen, sollte das klappen. Sehen wird uns dazu noch einmal die ersten Bildschirmadressen aus den ersten zwei Zeilen an.

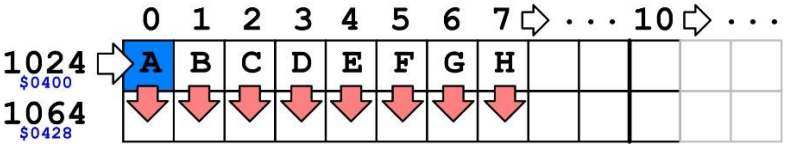


Abbildung 85 - Die ersten Bildschirmadressen der ersten beiden Zeilen

Ich habe für die ersten sieben Zeichen die Buchstaben ***A*** bis ***H*** eingesetzt, die auch später händisch eingegeben werden müssen. Im Anschluss soll das Programm gestartet werden, um zu sehen, was passiert. Es ist zu sehen, dass die Startadressen der beiden Bildschirmzeilen wie folgt lauten.

	Startadresse dezimal	**Startadresse hexadezimal**
Zeile 1	1024	$0400
Zeile 2	1064	$0428

Tabelle 11 - Die Startadressen der ersten beiden Bildschirmzeilen

Sehen wir uns das Programm für das Kopieren der ersten drei Zeichen der ersten Zeile in die zweite Zeile an. Ich komme im Anschluss zu den Erläuterungen.

Abbildung 86 - Das Programm für das Kopieren der Zeichen

Hier noch ein kleiner Hinweis zu SMON. nach der Eingabe des Assemblerprogramms und dem Abschluss über ***F*** habe ich den *Dump*-Befehl über die Eingabe von ***D*** und der Startadresse eingegeben. Im Anschluss wird die erste Zeile der angegebenen Adresse zur Anzeige gebracht. Um jede weitere Zeile anzuzeigen, muss solange einzeln die Leertaste gedrückt werden, bis die waagerechte Linie erscheint. Das Beenden des Dump-Modus wird über die *ESC*-Taste erreicht, so dass der blinkende Cursor wieder erscheint.

Doch nun zu den einzelnen Befehlen. Zu Beginn wird über den Befehl

A4000

die Startadresse angegeben. Nun folgen die Befehle. Der erste liest den Inhalt der angegebenen Speicherstelle *$0400* + den Inhalt des *X*-Registers in den Akku. Der Inhalt des *X*-Registers sollte zu Beginn *0* sein.

LDA $0400,X

Hier der OP-Code aus der Befehlsmatrix.

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC	ROR a	BBR6 r •	6
7	BVS r	ADC (zp),y	ADC (zp) *		STZ zp,x •	ADC zp,x	ROR zp,x	RMB7 zp •	SEI i	ADC a,y	PLY s •		JMP (a,x) *	ADC a,x	ROR a,x	BBR7 r •	7
8	BRA r •	STA (zp,x)			STY zp	STA zp	STX zp	SMB0 zp •	DEY i	BIT # *	TXA i		STY a	STA	STX a	BBS0 r •	8
9	BCC r	STA (zp),y	STA (zp) *		STY zp,x	STA zp,x	STX zp,y	SMB1 zp •	TYA i	STA a,y	TXS i		STZ a •	STA a,x	STZ a,x •	BBS1 r •	9
A	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp	SMB2 zp •	TAY i	LDA #	TAX i		LDY A	LDA	LDX a	BBS2 r •	A
B	BCS r	LDA (zp),y	LDA (zp) *		LDY zp,x	LDA zp,x	LDX zp,y	SMB3 zp •	CLV i	LDA A,y	TSX i		LDY a,x	LDA a,x	LDX a,y	BBS3 r •	B

Abbildung 87 - Die OP-Code-Matrixtabelle für den Op-Code LDA A,X

Der OP-Code für *LDA A,X* lautet *BD*, was auch im Assemblerprogramm zu sehen ist.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 12 - Die Flag-Beeinflussung des LDA-Befehls

Operation: M → A (Der Wert wird in den Akku übertragen)

Im nächsten Schritt wird der geladene Inhalt des Akkus in eine Adresse geschrieben, was über den *STA*-Befehl mit der Angabe der Speicheradresse und des *X*-Registers erfolgt.

STA $0428,X

Hier der OP-Code aus der Befehlsmatrix.

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r •	6
7	BVS r	ADC (zp),y	ADC (zp) *		STZ zp,x •	ADC zp,x	ROR zp,x	RMB7 zp •	SEI i	ADC a,y	PLY s •		JMP (a,x) *	ADC a,x	ROR a,x	BBR7 r •	7
8	BRA r •	STA (zp,x)			STY zp	STA zp	STX zp	SMB0 zp •	DEY i	BIT # *	TXA i		STY a	STA a	STX a	BBS0 r •	8
9	BCC r	STA (zp),y	STA (zp) *		STY zp,x	STA zp,x	STX zp,y	SMB1 zp •	TYA i	STA a,y	TXS i		STZ a •	STA a,x	STZ a,x •	BBS1 r •	9

Abbildung 88 - Die OP-Code-Matrixtabelle für den Op-Code STA A,X

Der OP-Code für *STA A,X* lautet *9D*, was auch im Assemblerprogramm zu sehen ist. Somit wäre der Kopiervorgang für ein einzelnes Zeichen abgeschlossen.

N	V	-	B	D	I	Z	C
-	-	-	-	-	-	-	-

Tabelle 13 - Die Flag-Beeinflussung des STA-Befehls

Operation: A → M (Der Inhalt des Akkus wird im Speicher abgelegt)

Ich möchte das aber für drei Zeichen durchführen, was bedeutet, dass die gezeigten Befehle noch zwei Mal wiederholt werden müssen, doch natürlich mit vorherigem Erhöhen, also Inkrementieren des *X*-Registers mit dem Befehl

INX

Hier der OP-Code aus der Befehlsmatrix.

E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX i	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ r	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 89 - Die OP-Code-Matrixtabelle für den Op-Code INX

Der OP-Code für *INX* lautet *E8*, was wiederum im Assemblerprogramm zu sehen ist. Das *i* steht für *immediate* und bedeutet unmittelbar.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 14 - Die Flag-Beeinflussung des INX-Befehls

Operation: X + 1 → X (Die Inkrementierung des X-Registers)

Um das Maschinenprogramm aus Basis heraus starten zu können, wird es mit dem Befehl

RTS

beendet. *RTS* steht für *Return from Subroutine* und bedeutet, dass nach der Beendigung des Maschinenprogramms, was in diesem Fall aus der Sicht von Basic ein Unterprogramm darstellt, die Kontrolle an Basic zurückgegeben wird. Hier der OP-Code aus der Befehlsmatrix.

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r •	6

Abbildung 90 - Die OP-Code-Matrixtabelle für den Op-Code RTS

Würden wir weiterhin den *BRK*-Befehl nutzen, würde nach dem Aufruf des Maschinenprogramms aus Basic heraus die Kontrolle im SMON-Monitor liegen.

N	V	-	B	D	I	Z	C
-	-	-	-	-	-	-	-

Tabelle 15 - Die Flag-Beeinflussung des RTS-Befehls

Bereiten wir also jetzt unseren Test vor. Hier die erforderlichen Schritte.

Schritt 1:

SMON mit der Eingabe von ***X*** verlassen.

Schritt 2:

Den kompletten Bildschirm mit der Tastenkombination in VICE mit ***SHIFT + POS1*** leeren.

Schritt 3:

In die erste Zeile die Buchstaben ***ABCDEFGH*** eingeben und nicht mit der *RETURN*-Taste bestätigen, denn ansonsten würde diese

Eingabe als Befehl interpretiert werden, was natürlich zu einem *Syntax Error* führen würde.

Schritt 4:

Mit den Cursortasten zum Beispiel in die 5. Zeile navigieren und dann den Befehl

SYS 16384

eingeben. Dieser dezimale Wert steht für die hexadezimale Zahl *$4000*, an der ja unser Assemblerprogramm seinen Anfang besitzt.

Abbildung 91 - Die Ausführung des Maschinenprogramms vorbereiten

Schritt 5:

Nun kann die Eingabe des *SYS*-Befehls mit der *RETURN*-Taste bestätigt werden. Die Anzeige gestaltet sich wie folgt.

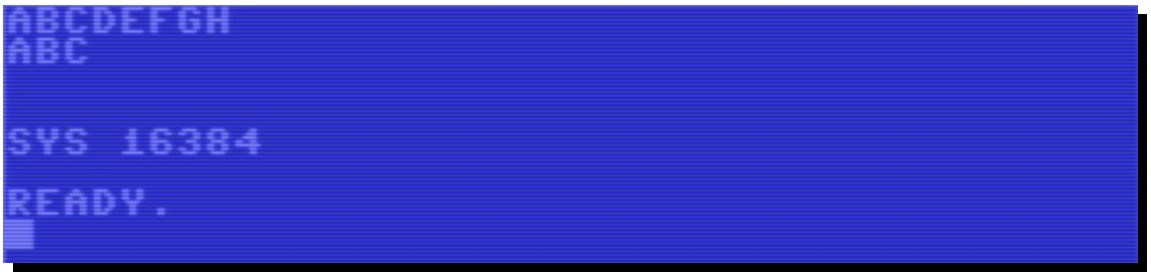


Abbildung 92 - Die Ausführung des Maschinenprogramms

Es ist zu sehen, dass die ersten drei Zeichen - das war ja auch vom Programm so vorgesehen - in die zweite Zeile direkt darunter kopiert wurden. Das sind die horizontalen Positionen *0*, *1* und *2*. Das Programm arbeitet perfekt. Doch ist das wirklich so? Leeren wir noch einmal den Bildschirm und führen die Prozedur erneut durch.

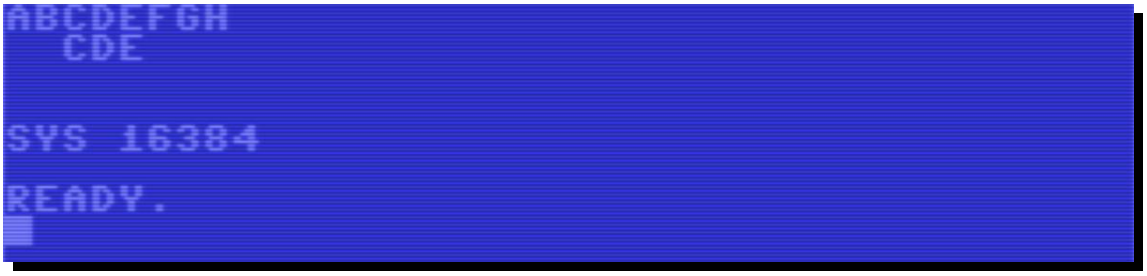


Abbildung 93 - Die erneute Ausführung des Maschinenprogramms

Ok, das Programm arbeitet noch, doch es sind jetzt nicht erneut die ersten drei Zeichen, sondern wieder drei Zeichen, doch ab Position *2*. Warum ist das so? Die Antwort liegt im unveränderten

X-Register, das nach der ersten Abarbeitung von *0* beginnend zwei Mal inkrementiert wurde. Mit diesem neuen Startwert *2* im *X*-Register arbeitet jetzt der erneute Aufruf des Programms. Wir haben also vergessen, den Wert des *X*-Registers vor der Abarbeitung der eigentlichen Kopierbefehle mit dem Wert *0* zu initialisieren. Rufen wir doch SMON erneut auf und sehen nach, welchen Inhalt das *X*-Register nach dem zweimaligen Aufruf des Programms vorweist.

Abbildung 94 - Das X-Register besitzt den Wert $04

Hier ist nun der Wert *$04* im *X*-Register zu sehen. Für jeden weiteren Aufruf des Programms würde es ohne die angesprochene Erweiterung in dieser Form weitergehen.

Das neue Programm schaut wie folgt aus.

Abbildung 95 - Das verbesserte Programm für das Kopieren der Zeichen

Hinzugekommen ist an erster Stelle der Befehl

LDX $#00

der das *X*-Register mit dem Wert *0* versieht. Hier der OP-Code aus der Befehlsmatrix.

A	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp	SMB2 zp •	TAY i	LDA #	TAX i		LDY A	LDA a	LDX a	BBS2 r •	A
B	BCS r	LDA (zp),y	LDA (zp) *		LDY zp,x	LDA zp,x	LDX zp,y	SMB3 zp •	CLV i	LDA A,y	TSX i		LDY a,x	LDA a,x	LDX a,y	BBS3 r •	B
C	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp	SMB4 zp •	INY i	CMP #	DEX i	WAI i •	CPY a	CMP a	DEC a	BBS4 r •	C
D	BNE r	CMP (zp),y	CMP (zp) *			CMP zp,x	DEC zp,x	SMB5 zp •	CLD i	CMP a,y	PHX s •	STP i •		CMP a,x	DEC a,x	BBS5 r •	D
E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX i	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ r	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED i	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 96 - Die OP-Code-Matrixtabelle für den Op-Code LDX

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 16 - Die Flag-Beeinflussung des LDX-Befehls

Operation: M → X (Ein Wert wird in das *X*-Register geladen)

Nun ist das Kopieren der einzelnen Zeichen auf diese Weise recht mühsam und eigentlich wollen wir ja die komplette erste Zeile in die zweite kopieren. Jetzt ist es natürlich möglich, die einzelnen Lade- und Speicherbefehle inklusive der Inkrementierung des *X*-Registers so viele Male im Code untereinander zu schreiben, bis die erste Zeile komplett erfasst und kopiert wurde. Doch es geht auch einfacher. Die Lösung liegt in der Implementierung einer Schleife, was im wiederholten Aufruf mehrerer Befehle besteht.

Die Indizierte Adressierung - Teil 2

Wie schon angekündigt, kommt nun der zweite Teil der indizierten Adressierung um die Ecke. Ich möchte das schon kurz geschilderte Problem mit der Adressierung aller möglichen Bildschirmpositionen von *$0400* (*1024*) bis *$07FF* (*2047*) vertiefen. Über die indizierte Adressierung können also eine Startadresse mit einem Offset von *255* möglichen Werten eine endgültige Adresse bilden. Also zum Beispiel.

- 1024 + 0
- 1024 + 1
- 1024 + 2
- ...
- 1024 + 255

Das sind in Summe 256 bildbare Adressen. Auf diese Weise kann der komplette Bildschirmbereich in vier Blöcke unterteilt werden, die je eine Startadresse besitzen und zum Beispiel über das *X*-Register indiziert werden.

Block	Startadresse	Endadresse
1	$0400 (1024)	$04FF (1279)
2	$0500 (1280)	$05FF (1535)
3	$0600 (1536)	$06FF (1792)
4	$0700 (1792)	$07FF (2047)

Tabelle 17 - Die vier Blöcke zur Bildschirmadressierung

Diese vier Blöcke können jetzt problemlos in ein Programm überführt werden, um alle *1000* Bildschirmpositionen zu erreichen. Schauen wir uns das genauer.

```
.D4000
,4000   A9 20      LDA #20
,4002   A2 00      LDX #00
,4004   9D 00 04   STA 0400,X
,4007   9D 00 05   STA 0500,X
,400A   9D 00 06   STA 0600,X
,400D   9D 00 07   STA 0700,X
,4010   CA         DEX
,4011   D0 F1      BNE 4004
,4013   00         BRK
--------------------------------
.
```

Abbildung 97 - Das Löschen des gesamten Bildschirms

Es werden also vier Blöcke zu je *256* Bytes angesprochen und mit dem Wert *#$20* versehen, was ein Leerzeichen ist. Hier ist übrigens ein schöner Vergleich hinsichtlich der Geschwindigkeit zwischen Basic und Maschinensprache zu erkennen. Natürlich können wir das Programm aus Basic heraus aufrufen, müssen dann aber den *BRK*-Befehl durch den *RTS*-Befehl ersetzen. Der Aufruf erfolgt dann mit

SYS 16384

Schreiben wir ein in der Funktion ähnliches Programm in BASIC, könnte das wie folgt aussehen, wobei der gesamte Bildschirm mit dem Buchstaben ***A*** gefüllt wird.

```
10 FOR I = 0 TO 999
20 POKE 1024 + I, 1
30 NEXT I
READY.
```

Abbildung 98 - Das Füllen des gesamten Bildschirms mit einem Buchstaben

Ok, nun sind wir soweit, dass ich zur nächsten Adressierungsart überleiten kann. Es handelt sich um die *indirekt-indizierte* Adressierung.

Die indirekt-Y-indizierte Adressierung

Das Problem mit der indizierten Adressierung ist derart, dass mit einem Offset von lediglich *0* bis *255* eine knappe Reichweite der Positionierung von nur *256* Bytes ermöglicht. Wenn wir irgendwie die Möglichkeit hätten, die Adresse zum Anfang des

Bildschirms sowohl im Low- als auch im High-Byte zu manipulieren, wären wir einen Schritt weiter. Angenommen, wir hätten diese Adresse, die auch Zeiger genannt wird, wie folgt in einem bestimmten Speicherbereich abgelegt. Sagen wir *$FA* (Low-Byte) und *$FB* (High-Byte). Diese beiden Adressen befinden sich in der Zero-Page.

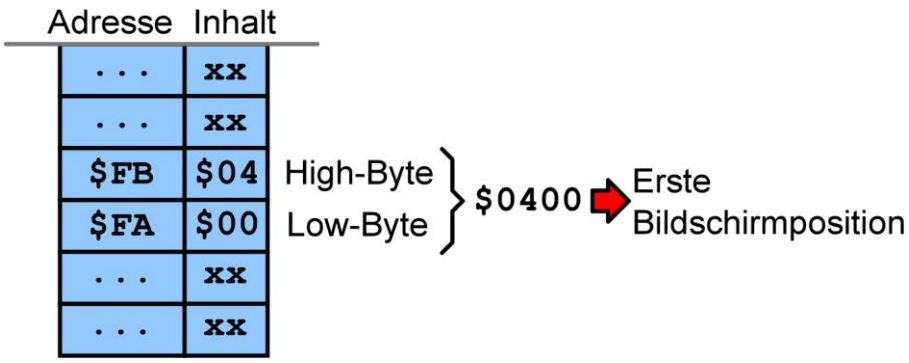


Abbildung 99 - Die Startadresse in der Zero-Page an $FA und $FB

Werfen wir noch einmal einen Blick auf die Speicherblöcke des Bildschirmspeichers, die ja nur virtuell so organisiert sind, um die Handhabung zu vereinfachen.

Block	Startadresse	Endadresse
1	$0400 (1024)	$04FF (1279)
2	$0500 (1280)	$05FF (1535)
3	$0600 (1536)	$06FF (1792)
4	$0700 (1792)	$07FF (2047)

Im ersten Block mit Staradresse *$0400* sind *256* Speicherstellen zu adressieren, was mit einem normalen Inkrementieren des Low-Bytes in *$FA* zum Beispiel mit dem *Y*-Register machbar ist. Also wie folgt.

High-Byte ($FB)	Low-Byte ($FA)
$04	$00
$04	$01
$04	$02
$04	$03
...	...

Genau diese *Y*-Register wird später bei der indirekt-indizierten Adressierung genutzt. Einen kleinen Augenblick noch. Sind diese *256* Schritte durchlaufen worden, kann es von vorne beginnen, doch diesmal erhöhen wir den Wert in der Speicherstelle *$FB*, die für das High-Byte steht.

Das Ganze würde dann wie folgt vonstattengehen.

High-Byte ($FB)	Low-Byte ($FA)
$05	$00
$05	$01
$05	$02
$05	$03
...	...

Und die Adresse *$0500* ist genau die Startadresse des zweiten Bildschirmblocks. So kann weiter verfahren werden, bis alle vier Bildschirmblöcke mit den je *256* Bytes angesprochen wurden.

Doch nun zur Grafik, die die indirekt-indizierte Adressierung verdeutlicht.

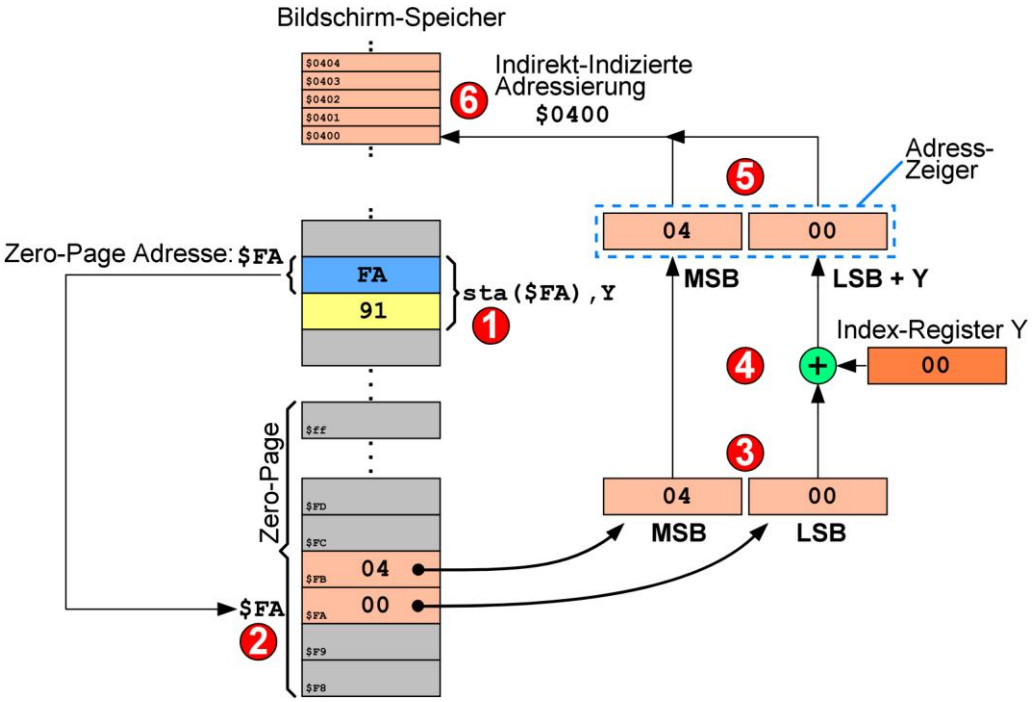


Abbildung 100 - Die indirekt-indizierte Adressierung

Der eigentliche Befehl an Punkt *1* lautet ja *STA ($FA),Y* und soll ja irgendetwas an eine bestimmte Speicherstelle schreiben.

Hier ein kurzer Einschub der Op-Code-Matrix für den neuen Befehl.

OP-Code: 91

9	BCC r	STA (zp),y	STA (zp) *		STY zp,x	STA zp,x	STX zp,y	SMB1 zp •	TYA i	STA a,y	TXS i		STZ a •	STA a,x	STZ a,x •	BBS1 r •	9
A	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp	SMB2 zp •	TAY i	LDA #	TAX i		LDY A	LDA a	LDX a	BBS2 r •	A
B	BCS r	LDA (zp),y	LDA (zp) *		LDY zp,x	LDA zp,x	LDX zp,y	SMB3 zp •	CLV i	LDA A,y	TSX i		LDY a,x	LDA a,x	LDX a,y	BBS3 r •	B
C	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp	SMB4 zp •	INY i	CMP #	DEX i	WAI i •	CPY a	CMP a	DEC a	BBS4 r •	C
D	BNE r	CMP (zp),y	CMP (zp) *			CMP zp,x	DEC zp,x	SMB5 zp •	CLD i	CMP a,y	PHX s •	STP i •		CMP a,x	DEC a,x	BBS5 r •	D
E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX i	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ r	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED i	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 101 - Die OP-Code-Matrixtabelle für den Op-Code STA (indirekt-indiziert)

Das ***(zp),y*** unterhalb des *STA*-Befehls bedeutet, dass es sich um einen Befehl handelt, der mit der Zero-Page arbeitet und zusätzlich das *Y*-Register zur Indizierung nutzt. Technisch gesehen wird dieser Befehl *Zero-Page-Indirect Y-Indexed* genannt.

Wie geht die Ereigniskette denn so weiter? Sehen wir uns das genauer an.

1. Über den Befehl *STA ($FB),Y* wird die indirekte Adressierung aufgrund der verwendeten Schreibweise mit den runden Klammern um die gezeigte Adresse eingeleitet.
2. Das bedeutet, dass aus der Speicherstelle *$FA* (Zero-Page) der darin gespeicherte Wert *$00* gelesen wird und als niederwertigster Adressteil (Low-Byte) Verwendung findet. Jetzt wird aber noch ein höherwertiger Teil (High-Byte) benötigt, der einfach aus der nächsten Speicherstelle *$FB* (also implizit) bezogen wird und in diesem Fall *$04* ist.
3. Somit haben wird schon einmal die Adresse *$0400* gebildet. Doch es geht noch weiter.
4. Zum Wert des LSB wird der Inhalt des *Y*-Registers - hier *$00* - hinzuaddiert.
5. Das Ergebnis der Addition der vorherigen Adresse *$0400* und dem Wert *$00* bildet den eigentlichen Adresszeiger mit dem Wert *$0400*.
6. Der Adresszeiger *$0400* weist natürlich auf eine bestimmte Stelle innerhalb des Speichers, was hier der Bildschirmspeicher ist.

Doch sehen wir uns jetzt das eigentliche Programm genauer an, das diese Funktionalität nutzt.

```
.D4000
,4000   A9 00      LDA #00
,4002   85 FA      STA   FA
,4004   A9 04      LDA #04
,4006   85 FB      STA   FB
,4008   A2 04      LDX #04
,400A   A0 00      LDY #00
,400C   A9 01      LDA #01
,400E   91 FA      STA  (FA),Y
,4010   88         DEY
,4011   D0 FB      BNE 400E
,4013   E6 FB      INC   FB
,4015   CA         DEX
,4016   D0 F6      BNE 400E
,4018   00         BRK
--------------------------------
.
```

Abbildung 102 - Die indirekt indizierte Adressierung

Das Programm wir nach der Ausführung den kompletten Bildschirm mit dem Buchstaben ***A*** füllen. Nach dem Start über

G4000

schaut es schon etwas wild auf dem Bildschirm aus.

Abbildung 103 - Das Programm wurde ausgeführt

Ich schlage vor, den *BRK*-Befehl wieder durch *RTS* zu ersetzen und dann von Basic aus, das Programm über

SYS 16384

zu starten. Aber schauen wir uns das Programm etwas genauer an. Auf der folgenden Abbildung ist zu sehen, wie über die Speicherung der Werte *$00* nach *$FA* (Low-Byte) und *$04* nach *$FB* (High-Byte) der Zeiger für die spätere indirekte-indizierte Adressierung kopiert werden.

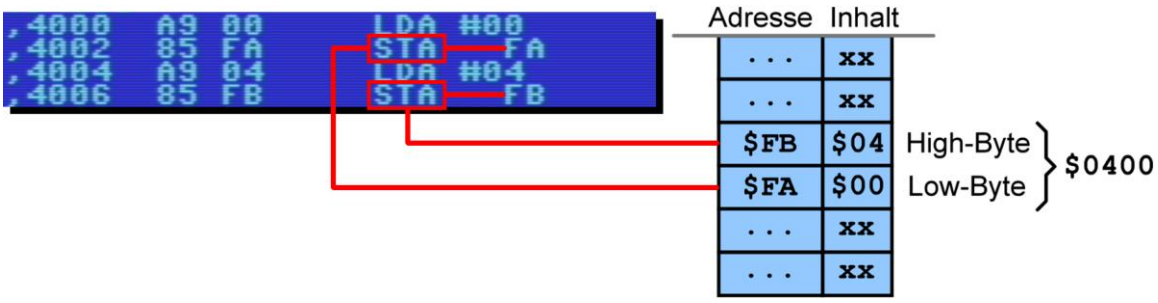


Abbildung 104 - Die Basisadresse wird in der Zero-Page gesetzt

Nachfolgend kommt es zu Initialisierung von Werten für die erforderlichen Schleifendurchläufe und dem anzuzeigenden Zeichen.

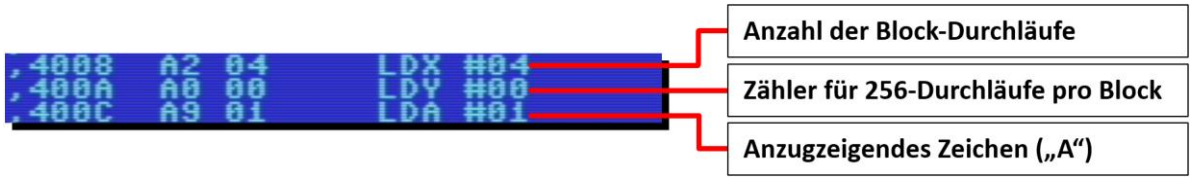


Abbildung 105 - Die Initialisierung von Ausgangswerten

Entscheidend für die indirekte-indizierte Adressierung ist die nachfolgende Zeile mit der besonderen Schreibweise über die runden Klammern.

```
,400E  91 FA      STA  (FA),Y
```

Der *STA*-Befehl schaut nun nach, welcher Wert sich hinter der Adresse *$FA* verbirgt und nutzt diesen als das Low-Byte für die Startadresse respektive Bildschirmadresse. Implizit ist hier klar, dass die darauffolgende Adresse *$FB* das High-Byte für selbige Adresse beinhaltet. Zusätzlich kommt nun noch der Offset hinzu, der sich im *Y*-Register befindet. Dieser drei Einzelkomponenten bilden in Summe die Zieladresse. In der nachfolgend zu sehenden Schleife werden die *256* Zeichen je Block abgearbeitet, wobei das *Y*-Register in diesem Fall als Zähler für den genannten Offset-Wert arbeitet. Wie ein Zähler im Detail funktioniert und was dabei berücksichtigt wird, kommt im nachfolgenden Kapitel „Wir zählen“ zur Sprache. Also ein wenig Geduld noch. Für das Verständnis an dieser Stelle sollte das jedoch ausreichend sein.

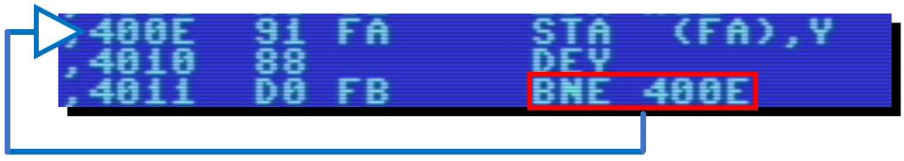


Abbildung 106 - Der Zähler für die 256 Zeichen pro Block

Wurde diese Schleife komplett abgearbeitet, muss das gleich für den nächsten Block erfolgen. Das wird darüber erreicht, das High-

Byte an Speicheradresse *$FB* über den *INC*-Befehl zu inkrementieren.

```
.4013  E6 FB     INC  FB
```

Abbildung 107 - Das High-Byte wird inkrementiert

Zu guter Letzt wird über den *DEX*-Befehl in Adresse *$4015* das *X*-Register dekrementiert, das ja für die Anzahl der Block-Durchläufe zuständig ist. Falls die Bedingung für das Schleifenende noch nicht erreicht wurde, beginnt der Vorgang erneut durch einen Sprung an Adresse *$400E* für einen erneuten Durchlauf der nächsten *256* Bytes je Block.

```
.400E  91 FA     STA  (FA),Y
.4010  88        DEY
.4011  D0 FB     BNE  400E
.4013  E6 FB     INC  FB
.4015  CA        DEX
.4016  D0 F6     BNE  400E
```

Abbildung 108 - Sind alle vier Blöcke abgearbeitet?

Wurden alle vier Blöcke abgearbeitet, kommt es mit der nachfolgenden *BRK*-Anweisung zu einer Programmunterbrechung.

```
.4018  00        BRK
```

Abbildung 109 - Die abschließende Programmunterbrechung

Somit ist das Programm erfolgreich abgearbeitet worden. Abschließend können wir sagen, dass über die *indirekt-indizierte Adressierung* es nun möglich ist, verschiedene Speicherbereiche durchlaufen zu lassen, alleine, durch Verändern der vermeintlich festen Anfangsadresse. Aufgrund der Möglichkeit der Indizierung bietet sich diese Methode sehr gut zur Adressierung hintereinanderstehender Werte im Speicher, die die Charakteristik einer Tabelle besitzen. Die Einschränkung besteht natürlich in der schon genannten Tabellenlänge von *256* Elementen respektive Speicherstellen, die mit einer Datenbreite von *8* Bits in einem Schwung zu erreichen sind.

Die indirekt-X-indizierte Adressierung

Bei dieser Adressierungsart steht das *X*-Registers im Vordergrund und zeigt im Zusammenspiel mit dem angegeben Operand in Summe auf einen Speicherbereich innerhalb der Zero-Page, also zwischen *$0000* und *$00FF* und indiziert diese indirekte Adressierung. Die eigentliche Speicheradresse, die es gilt auszuwählen, wird durch *2* Bytes in der Zero-Page gebildet, wobei das LSB durch den Operanden *+ X* gebildet wird und das MSB in der nächsthöheren Speicheradresse zu finden ist. Das Ganze ist am besten wieder in einem schönen bunten Diagramm zu sehen.

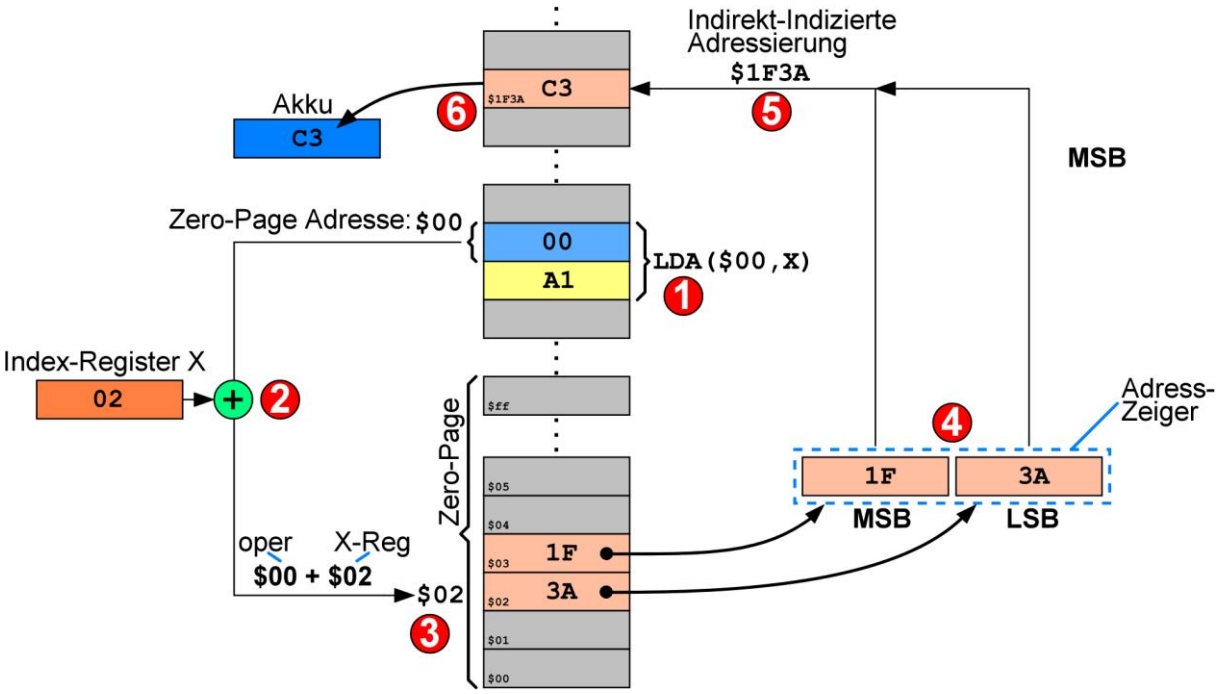


Abbildung 110 - Die X-indiziert indirekte Adressierung

Der besseren Übersicht wegen, habe ich die wichtigen Wegepunkte einer Nummerierung von *1* bis *6* unterzogen.

1. Über den Befehl *LDA($00,X)* wird die indirekte Adressierung aufgrund der verwendeten Schreibweise mit den runden Klammern um die gezeigte Adresse und *X*-Register eingeleitet.
2. Das *X*-Register ist zur Offset-Bildung mit dem Wert *$02* geladen.
3. Das bedeutet im weiteren Verlauf, dass aus der Summierung von Startadresse *$00* und *X*-Register *$02* die Speicherstelle *$02* in der Zero-Page ausgewählt wird und der dort zu findende Wert als niederwertigster Adressteil LSB Verwendung findet. Jetzt wird aber noch ein höherwertiger Teil MSB benötigt, der einfach aus der nächsten Speicherstelle *$03* (also implizit) bezogen wird und in diesem Fall *$1F* ist.
4. Somit wurde der benötigte Adresszeiger aus dem LSB und MSB der Zero-Page gebildet, die nun *$1F3A* lautet.
5. Dieser Adress-Zeiger *$1F3A* zeigt auf eine Speicheradresse, in der der Wert *$Cc3* zu finden ist.
6. Der Wert *$C3* wird letztendlich in den Akku zur weiteren Verarbeitung geladen.

Die relative Adressierung

Bei der relativen Adressierung kommt es zu recht kurzen Sprüngen relativ zum Inhalt des aktuellen Programmzählers.

	Die relative Adressierung
Die relative Adressierung gibt es nur bei Sprungbefehlen (Branches). Der relative Sprung kann dabei maximal *-128* (Sprünge zurück) oder *+127* (Sprünge nach vorne) Bytes um den Befehl reichen.	

Wir zählen

Wenn es darum geht, einen oder mehrere Befehle wiederholt zur Ausführung zu bringen, benötigen wir zwei Konstrukte, die diese Aufgabe erfüllen können. In erster Instanz ist da ein Zähler, der rauf oder runterzählen kann und in zweiter Instanz eine Bewertungsinstanz, die Entscheidungen hinsichtlich eines vorliegenden Wertes treffen kann. Sehen wir uns zunächst den Zähler an. Hier stehen die unterschiedlichsten Befehle zur Verfügung. Zum einen gibt es können Register manipuliert werden und zum anderen Speicherstellen.

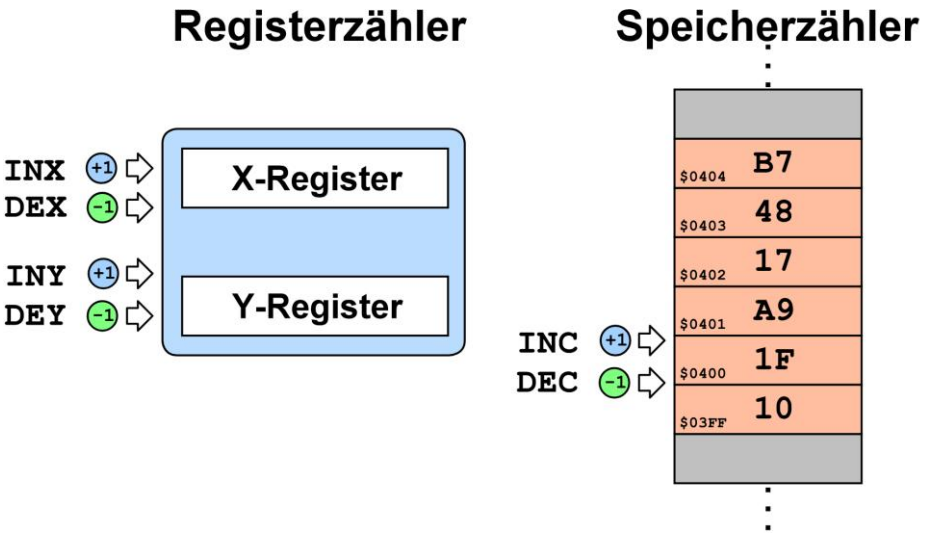


Abbildung 111 - Befehle zur Register- und Speichermanipulation

Auf dieser Abbildung sind Befehle zu sehen, die das *X*-Register, das *Y*-Register und eine beliebige Speicheradresse um den Wert *1* erhöhen (inkrementieren) oder um den Wert *1* erniedrigen (dekrementieren) können.

- **INX: IN**crement **X**-Register (+1)
- **DEX: DE**crement **X**-Register (-1)
- **INY: IN**crement **Y**-Register (+1)
- **DEY: DE**crement **Y**-Register (-1)

- **INC: INC**rement Memory (+1)
- **DEC: DEC**rement Memory (-1)

Es handelt sich bei den Registerbefehlen (*X*- und *Y*-Register) um *1* Byte-Befehle, was einer impliziten Adressierung gleichkommt. Bei der Nutzung der Speicheradressenmanipulation muss natürlich ein Speicheradresse angegeben werden, die entweder 8-Bit (Zero-Page) oder 16-Bit (größer *$FF*) breit sein kann. Dann handelt es sich um einen *2*-Byte- oder *3*-Byte-Befehl). Entsprechende Befehle zur Manipulation des Akkus existieren leider nicht.

Es versteht sich, dass die Zähler lediglich von *0* bis *255* oder umgekehrt zählen können. Wird ein Wert, der *0* war erniedrigt, ist das Ergebnis *255*. Ebenso wird ein Wert, der *255* war erhöht, ist das Ergebnis *0*.

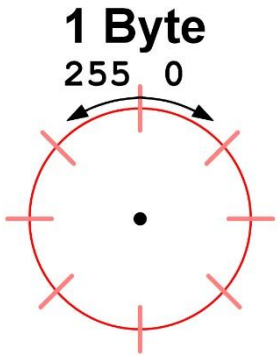


Abbildung 112 - Der 1-Byte-Zählerkreis

Kommen wir jetzt zum zweiten genannten Punkt, der über die Möglichkeit einer Entscheidung zu treffen verfügt. Es handelt sich dabei um die sogenannten *bedingten Anweisungen*. Wie der Name schon vermuten lässt, spielt hier eine Bedingung eine entscheidende Rolle. Damit ist gemeint, dass bestimmte Befehle nur dann zur Ausführung kommen, wenn eine vorher definierte Bedingung erfüllt ist. Aus der Sprache Basic sind das die *Wenn-Dann-Konstrukte*, die mit *IF-THEN* programmiert werden, wie das bei dem folgenden Beispiel zu sehen ist.

```
IF A=10 THEN 100
```

Erst, wenn der Wert der Variablen *A* gleich *10* ist, wird zur Ausführung der Befehle ab Zeile *100* begonnen. Falls dem nicht so ist, wird mit der darauffolgenden Zeile des Programms fortgefahren. In Assembler arbeiten hier meistens zwei Befehle Hand in Hand. Zum einen ist das ein Befehl, der den Vergleich mit einem bestimmten Wert durchführt und darüber das Statusregister mit seinen Flags modifiziert wird. Zum anderen ist dann ein Befehl erforderlich, der bestimmte Flags auswertet und in Abhängigkeit des Ergebnisses entweder einen Sprung zu einer bestimmten Speicheradresse durchführt oder eben nicht. Auf diese Weise können sogenannte Schleifen realisiert werden.

Was ist eine Schleife?

Eine Schleife - englisch loop genannt - ist eine Kontrollstruktur in unterschiedlichen Programmiersprachen. Sie wird immer dann eingesetzt, um bestimmte Anweisungen-den sogenannten Schleifenrumpf - solange auszuführen, wie die Schleifenbedingung logisch *wahr* bleibt oder eine Abbruchbedingung eintritt.

Auf der folgenden Abbildung ist eine einfache Schleife über ein Flussdiagramm zu sehen.

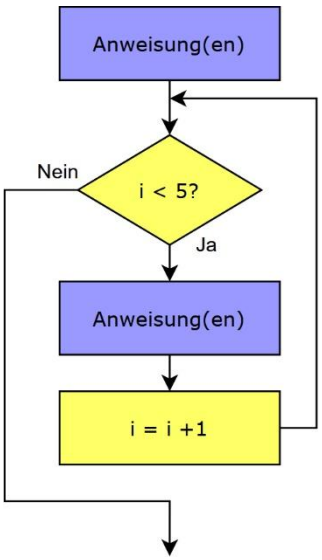


Abbildung 113 - Eine einfache Schleife

Erreicht die Programmausführung die Raute (*i < 5?*), in der eine Bedingung formuliert ist, findet eine Bewertung statt. Ist der Inhalt der Variablen kleiner dem Wert *5*, erfolgt die Fortführung gemäß dem Pfad, der mit *Ja* beschriftet ist. Die darunter liegenden Anweisungen werden ausgeführt. Damit es nicht zu einer Endlosschleife kommt, muss die Variable, die in der Bedingung vorkommt, in irgendeiner Form verändert werden. Das passiert hinter der Anweisung, wo *i = i + 1* steht, was bedeutet, dass eine Inkrementierung der Variablen *i* um den Wert *1* erfolgt. Anschließend wird wieder zur Bewertung verzweigt, um dort zu erneut prüfen, ob der Inhalt der Variablen immer noch kleiner *5* ist. Ist das nicht der Fall, ist quasi eine Abbruchbedingung der Schleife erreicht, wird mit der Programmausführung über den Pfad fortgeführt, der mit *Nein* gekennzeichnet ist. Wird die Schleife mit dem Startwert *i = 1* begonnen, kommt es zu einem *4*-maligen Durchlauf der Schleife. Für einen *5*-maligen Durchlauf muss entweder der Startwert von *i* auf *0* gesetzt werden oder die Bedingung auf *i <= 5* angepasst werden.

Zählervariante 1

Ich möchte das anhand eines kleinen Programms verdeutlichen, dass eine bestimmte Anzahl des Buchstabens ***A*** auf dem Bildschirm anzeigt. Sagen wir einmal, es sollen *8* Buchstaben angezeigt werden. Als Schleifenzähler nutze ich dabei das *Y*-Register. Lasse dich nicht durch die neuen Befehle ***CPY*** und ***BNE*** verwirren, denn ich komme sofort darauf zu sprechen.

Abbildung 114 - Die Programmierung einer Schleife

Gehen wir das Ganze der Reihe nach durch.

Schritt 1 (ab Speicherstelle $4000):

Es wird der Akku mit dem Wert *#$01* geladen, um später den Buchstaben ***A*** anzuzeigen.

Schritt 2 (ab Speicherstelle $4002):

Es wird das *Y*-Register mit dem Wert *#$00* geladen, um das Register als Zähler zu nutzen und diesen später schrittweise zu inkrementieren.

Schritt 3 (ab Speicherstelle $4004):

Es wird der Inhalt des Akkus über die indizierte Adressierung mit dem *Y*-Register an die Speicherstelle *$0400+Y* geschrieben.

Schritt 4 (ab Speicherstelle $4007):

Über den *INY*-Befehl wird das *Y*-Register inkrementiert. Sehen wir uns den entsprechenden OP-Code zum *INY*-Befehl an.

C	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp	SMB4 zp •	INY i	CMP #	DEX i	WAI i •	CPY a	CMP a	DEC a	BBS4 r •	C
D	BNE r	CMP (zp),y	CMP (zp) *			CMP zp,x	DEC zp,x	SMB5 zp •	CLD	CMP a,y	PHX s •	STP i •		CMP a,x	DEC a,x	BBS5 r •	D
E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ r	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 115 - Die OP-Code-Matrixtabelle für den Op-Code INY

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 18 - Die Flag-Beeinflussung des LDY-Befehls

Operation: M → Y (Ein Wert wird in das *Y*-Register geladen)

Schritt 5 (ab Speicherstelle $4008):

Über den *CPY*-Befehl mit dem Operanden *#$08* wird ein Vergleich durchgeführt. CPY steht für ***C**om**P**are **Y**-Register* und bedeutet übersetzt: „Vergleiche das *Y*-Register mit dem Wert *#$08*." Jedoch beinhaltet dieser Befehl keine weiteren Informationen, wie dieser Vergleich zu bewerten ist. Also ob auf Gleichheit, Ungleichheit, Kleiner oder Größer untersucht werden soll. Sehen wir uns den entsprechenden OP-Code zum *CPY*-Befehl an.

C	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp	SMB4 zp •	INY i	CMP #	DEX i	WAI I •	CPY a	CMP a	DEC a	BBS4 r •	C
D	BNE	CMP (zp),y	CMP (zp) *			CMP zp,x	DEC zp,x	SMB5 zp •	CLD i	CMP a,y	PHX s •	STP I •		CMP a,x	DEC a,x	BBS5 r •	D
E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX i	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED i	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 116 - Die OP-Code-Matrixtabelle für den Op-Code CPY

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	✓

Tabelle 19 - Die Flag-Beeinflussung des CPY-Befehls

Operation: Y - M (Die Differenzbildung von Y-Register und Wert)

Dieser Befehl führt eine Subtraktion im Zweierkomplement zwischen dem *Y*-Register und der angegebenen Speicherstelle durch. Das Ergebnis der Subtraktion wird nicht gespeichert, sondern ausschließlich zur Beeinflussung der Flags verwendet.

Jetzt kommt der nächste Befehl ins Spiel.

Schritt 6 (ab Speicherstelle $400A):

Über den *BNE*-Befehl, der übersetzt ***B**ranch if **N**ot **E**qual* lautet, wird eine mögliche Verzweigung, also ein bedingter Sprung definiert. Es soll nur dann zur angegebenen Adresse gesprungen werden, wenn die verglichenen Werte nicht gleich sind. In unserem Fall lautet das Zusammenspiel zwischen *CPY* und *BNE* konkret formuliert wie folgt: „Vergleiche das Y-Register mit dem Wert *#$08* (CPY #$08) und springe bei Ungleichheit zur Adresse *$4004* (BNE $4004). Wurde die Bedingung auf Ungleichheit nicht erfüllt, erfolgt kein Sprung zur angegebenen Adresse *$4004* und es wird der nachfolgende Befehl *RTS* ausgeführt. Doch stopp einmal! Hier

stimmt doch etwas nicht. Wir sehen uns noch einmal die entsprechende Befehlszeile an, in der der *BNE*-Befehl steht.

Abbildung 117 - Der bedingte Rückwärtssprung zur Adresse $4004

Sehen wir uns den entsprechenden OP-Code zum *BNE*-Befehl an.

D	BNE r	CMP (zp),y	CMP (zp) *			CMP zp,x	DEC zp,x	SMB5 zp •	CLD i	CMP a,y	PHX s •	STP I •		CMP a,x	DEC a,x	BBS5 r •	D
E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX i	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED i	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 118 - Die OP-Code-Matrixtabelle für den Op-Code BNE

Der OP-Code für *BNE* ist *D0* und es steht ein ***r*** dort unterhalb des Befehls. Der Buchstabe steht als Abkürzung für eine relative Adressierung.

N	V	-	B	D	I	Z	C
-	-	-	-	-	-	-	-

Tabelle 20 - Die Flag-Beeinflussung des BNE-Befehls

Operation: Verzweige, wenn Z-Flag = 0

In unserem Assemblerprogramm steht hinter dem OP-Code *D0* der Wert *F8* und nicht etwa die angegebene Sprungadresse *$4004*.

D0 F8

Man würde doch erwarten, dass dort vielleicht

D0 04 40

steht. Wie passt das also mit der angesprochenen relativen Adressierung zusammen? Es steht für die Sprungadresse anscheinend nur ein einziges Byte zur Verfügung. Man muss das Byte mit dem Wert *$F8* als Offset-Adresse (Sprungweite) zum aktuellen Programmzähler sehen, der schon auf den nächsten Befehl zeigt. Mit einem Byte sind 256 unterschiedliche Werte abzubilden und man sollte meinen, dass es möglich ist, von der aktuellen Speicherposition *256* Schritte in eine Richtung zu unternehmen, was aber falsch ist, denn es müssen ja beide Richtungen quasi bedient werden. Also sowohl Sprünge nach vorne,

als auch Sprünge zurück. Bevor ich zum nächsten Schritt übergehe, sollten wir uns das genauer ansehen. Dazu habe ich das gerade verwendete Programm im Speicher dargestellt. Die niedrigste Adresse *$0400* befindet sich diesmal am oberen Ende der Grafik.

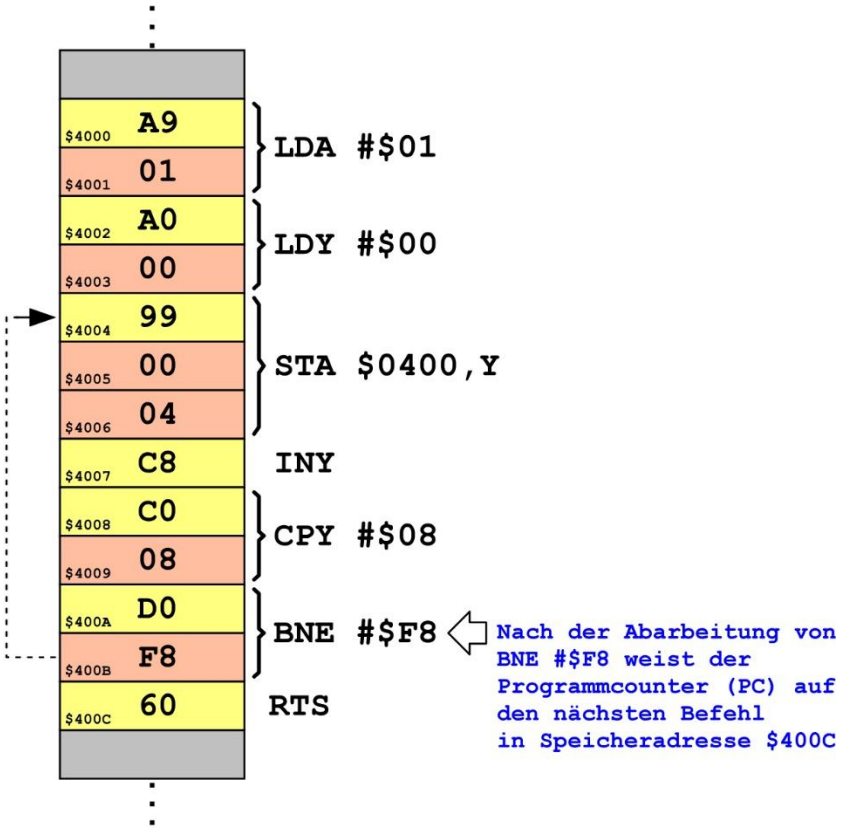


Abbildung 119 - Das Programm im Speicher

Es ist wichtig zu erkennen, dass der Programmzähler (PC) nach der Abarbeitung eines Befehls immer auf die nächste abzuarbeitende Speicherstelle weist, was im Falle von *BNE #$F8* die Adresse *$400C* ist. Von dort auf muss jetzt ein Rücksprung zur Adresse *$4004* erfolgen, um den nächsten Schleifendurchlauf zu gewährleisten. Was ist die Differenz von *$400C* und *$4004*? Nun, das ist... Doch bemühen wir zur Berechnung der Differenz SMON. Wir wissen jetzt, wie das geht.

```
.?400C-4004
  08 00001000   8
.
```

Abbildung 120 - SMON berechnet die Differenz von $400C und $4004

Das Ergebnis ist *$08*. Der Programmzähler muss also acht Positionen zurückgesetzt werden, um beim Start der Schleifenausführung an Adresse *$4004* anzugelangen. Wir haben aber gesehen, dass positive Werte für relative Adressierung für Vorwärtsschritte genutzt werden. Damit aber Rückschritte bewirkt werden, muss es sich um einen negativen Wert handeln. Und das genau hängt mit dem Operanden *$F8* hinter dem *BNE*-Befehl zusammen.

Aber warum sollte das ein negativer Wert sein? In einem einzigen Byte können doch Werte von *0* bis *255* respektive *$00* bis *$FF* gespeichert werden und das sind allesamt positive Werte. Ich sprach zu Beginn schon einmal davon, dass das eine Interpretationssache ist. Der Wert *$F8* (dezimal 248) schaut binär wie folgt aus. Das Negative-Flag (Bit 7 - MSB) ist gesetzt, was bedeutet, dass ein relativer Rücksprung erfolgen muss. Wie weit das ist, muss jetzt über die Bildung des Zweierkomplements errechnet werden.

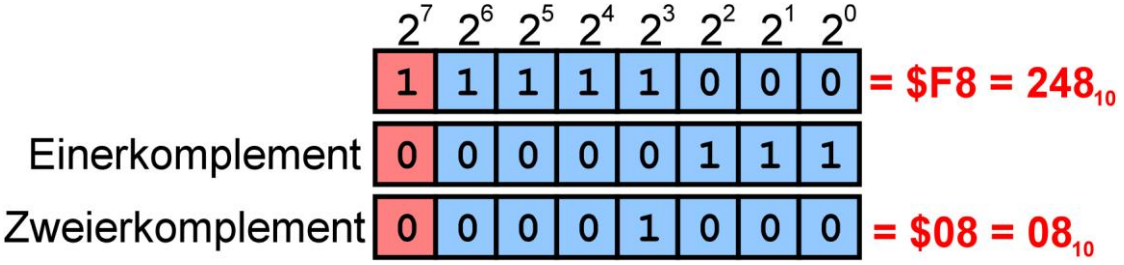


Abbildung 121 - Die Berechnung des Zweierkomplements von $F8

Wir können sehen, dass es sich um acht Schritte handelt, die von der aktuellen Programmzähleradresse zurückgerechnet werden müssen. Somit sollte klar sein, wie die relative Adressierung für einen Rücksprung funktioniert. Man kann sich das übrigens sehr einfach merken, in dem man in diesem Fall von *$FF* rückwärts zählt.

Operand	$FF	$FE	$FD	$FC	$FB	$FA	$F9	$F8	$F7	...
Offset	1	2	3	4	5	6	7	8	9	...

Gleich kommen wir noch zu einem Beispiel für einen Vorwärtssprung. Doch zuerst einmal zurück zum nächsten Schritt.

Schritt 7 (ab Speicherstelle $400C):

Über den *RTS*-Befehl wird das Unterprogramm verlassen, wenn die Schleife verlassen wurde. Sehen wir uns abschließend dazu die Ausführung des Programms an.

Abbildung 122 - Die Ausführung des Schleifenprogramms

Es sind acht ***A**'s* in der ersten Bildschirmzeile zu sehen. Ich würde jetzt ein wenig mit unterschiedlichen Werten sowohl des *Y*-Registers, als auch des *CPY*-Befehls experimentieren. Es ist sinnvoll, vor jedem neuen Test die Buchstaben zu entfernen, denn ansonsten können die Ergebnisse nicht korrekt erkannt werden.

Oder du modifizierst den Code derart, dass vor dem Schreiben der Buchstaben die erste Zeile komplett gelöscht wird. Dir Grundlagen dafür sind bekannt. In diesem Beispiel haben wir also gesehen, wie in einem Programm mit Branch-Befehlen Sprungadressen eingegeben werden müssen. Beim Programmieren ist es jedoch meisten nicht sofort ersichtlich, wie die Zieladressen genau lauten und deshalb gibt es eine Möglichkeit mit Labels zu arbeiten. Das ist zwar - ich erwähnte es schon zu Beginn - nicht so komfortabel wie bei einem Assemblerprogramm wie zum Beispiel dem *TASM 7.4*, doch eine Erleichterung ist es allemal. In SMON können zu diesem Zweck Labels in Form von vordefinierten Marken in der Schreibweise ***Mxx*** im Bereich von *01* bis *30* eingegeben werden, wobei es sich bei den Werten um hexadezimale Zahlen handelt. Die Eingabe von

M01 STA 0400,X

markiert die entsprechende Startadresse, in der sich der Befehl befindet. Sehen wir uns das genauer am gerade erstellen Programm an.

Abbildung 123 - Das Label M01 wird gesetzt

In der letzten Zeile habe ich also vor der Eingabe des eigentlichen Befehls das Label ***M01*** platziert. Drücke ich jetzt die *RETURN*-Taste, merkt sich SMON im Hintergrund die Adresse $4004 und die nachfolgende Anzeige ist zu sehen.

Abbildung 124 - Das Label M01 wurde im Hintergrund gespeichert

Vom eigentlichen *M01*-Label ist hier nichts mehr zu sehen. Natürlich verwaltet SMON alle eingegebenen Labels intern und wartet, bis ein Aufruf des Labels bei einem Branch-Befehl auftaucht, um dann einzugreifen, wie das beim *BNE*-Befehl jetzt der Fall ist. Hinter dem befehl muss nun nicht die Adresse für den Start des nächsten Schleifendurchlaufs eingegeben werden, die $4004 lautet, sondern einfach die Marke M01.

Abbildung 125 - Die Marke M01 wird eingegeben

Nach der Bestätigung über die *RETURN*-Taste liefert SMON die folgende Zeile zurück.

```
400A  D3 01      BNE M01
400C
```

Abbildung 126 - Die Marke M01 wurde übernommen

Noch ist hier die eigentliche Sprungadresse zu sehen. Beenden wir jetzt die Eingabe des Programms mit dem *RTS*-Befehl und der nachfolgenden Eingabe von *F* mit der Bestätigung über die *RETURN*-Taste.

```
 400D F
,4000  A9 01     LDA #01
,4002  A0 00     LDY #00
,4004  99 00 04  STA 0400,Y
,4007  C8        INY
,4008  C0 08     CPY #08
,400A  D3 01     BNE 4004
,400C  60        RTS
--------------------------------
.
```

Abbildung 127 - Die Marke M01 wurde durch die entsprechende Adresse ersetzt

Es ist zu sehen, dass die *M01*-Marke durch die korrekte Adresse *$4004* ersetzt wurde.

Zählervariante 2

Wie versprochen, möchte ich jetzt ein Zählerprogramm zeigen, bei dem ein positiver Wert für eine relative Adressierung vorkommt. Werfen wir dazu einen Blick auf das folgende Programm.

```
  PC   SR AC XR YR SP   NV-BDIZC
;C00B  B0 C2 00 00 EF   10110000
.D4000
,4000  A9 01     LDA #01
,4002  A2 00     LDX #00
,4004  9D 00 04  STA 0400,X
,4007  E8        INX
,4008  F0 03     BEQ 400D
,400A  4C 04 40  JMP 4004
--------------------------------
,400D  60        RTS
--------------------------------
.
```

Abbildung 128 - Die Programmierung einer Schleife

In diesem Programm sind zwei neue Befehle enthalten. Lasse dich also nicht durch die neuen Befehle ***BEQ*** und ***JMP*** verwirren, denn ich komme sofort darauf zu sprechen.

Doch gehen wir wieder schrittweise vor. Das Programm soll 256 Zeichen in Form des Buchstabens ***A*** in den Bildschirmspeicher schreiben, was wiederum nichts Spektakuläres ist, doch auf Ästhetik kommt es nicht an.

Schritt 1 (ab Speicherstelle $4000):

Es wird der Akku mit dem Wert *#$01* geladen, um später den Buchstaben ***A*** anzuzeigen.

Schritt 2 (ab Speicherstelle $4002):

Es wird das *X*-Register mit dem Wert *#$00* geladen, um das Register als Zähler zu nutzen und diesen später schrittweise zu inkrementieren.

Schritt 3 (ab Speicherstelle $4004):

Es wird der Inhalt des Akkus über die indizierte Adressierung mit dem *X*-Register an die Speicherstelle *$0400+X* geschrieben.

Schritt 4 (ab Speicherstelle $4007):

Über den *INX*-Befehl wird das *X*-Register inkrementiert. Dieser Befehl beeinflusst die nachfolgenden Flags.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Für uns wichtig ist wieder das Z-Flag (Zero-Flag), was später vom Branch-Befehl *BEQ* ausgewertet wird.

Schritt 5 (ab Speicherstelle $4008):

Über den neuen *BEQ*-Befehl, der übersetzt ***B****ranch On* ***EQ****ual* lautet, wird eine mögliche Verzweigung, also ein bedingter Sprung definiert. Es wird eine bedingte Verzweigung durchgeführt, wenn das *Z*-Flag gesetzt ist oder das vorherige Ergebnis gleich 0 ist. Bei diesem Programm erfolgt kein Vergleich mit einem bestimmten Wert, sondern lediglich der fortlaufende Aufruf des *INX*-Befehls, der aber natürlich auch das *Z*-Flag beeinflusst.

F	BEQ r	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED i	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 129 - Die OP-Code-Matrixtabelle für den Op-Code BEQ

Kommt es beim Hochzählen von *0* an zu einem Überlauf, wird also der Wert des *X*-Registers nach *255* wieder *0*, wird das *Z*-Flag gesetzt. Aus diesem Grund wir der Schleifendurchlauf auch *256*-mal erfolgen. Kommen wir nun wieder auf die relative Adressierung zu sprechen und werfen einen Blick auf den *BEQ*-Befehl.

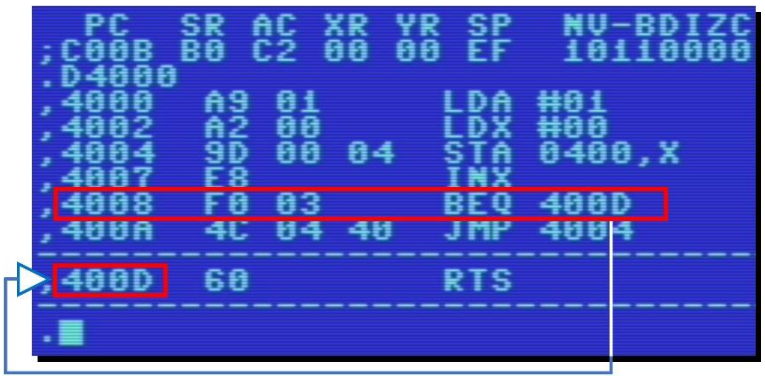


Abbildung 130 - Der bedingte Vorwärtssprung zur Adresse $400D

Der Operand des *BEQ*-Befehls lautet in diesem Fall *$03* und wenn man sich den Wert in binärer Schreibweise (*00000011*) anschaut, dann sieht man sofort, dass das MSB (Bit 7) nicht gesetzt ist und deshalb dieser Wert als Vorwärtssprung identifiziert wird. Schauen wir uns das noch einmal genauer an.

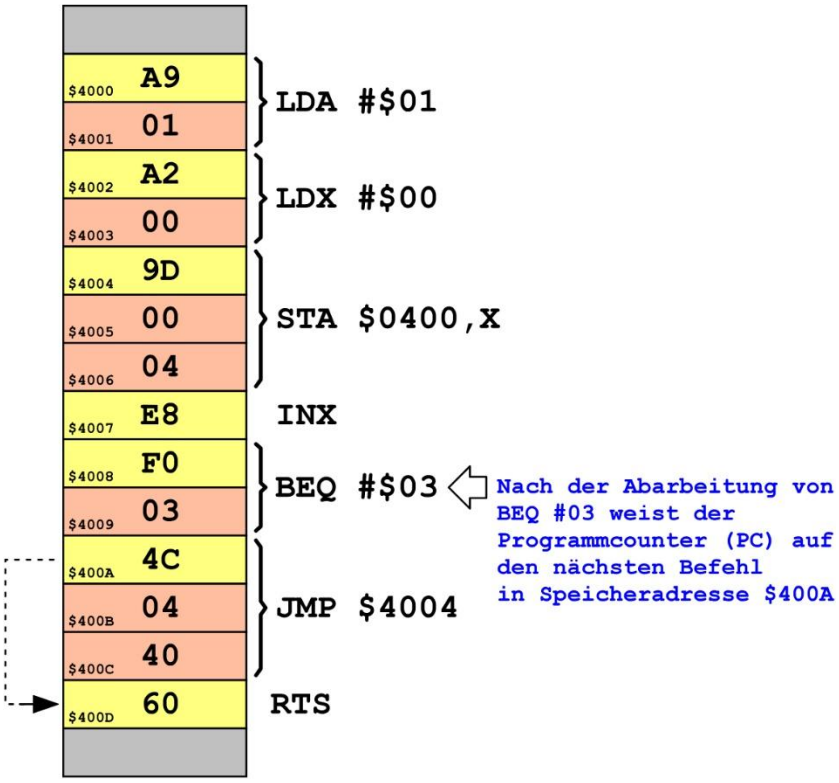


Abbildung 131 - Das Programm im Speicher

Es ist wiederum wichtig zu erkennen, dass der Programmzähler (PC) nach der Abarbeitung eines Befehls immer auf die nächste abzuarbeitende Speicherstelle weist, was im Falle von *BEQ #$03* die Adresse *$400A* ist. Von dort auf muss jetzt ein Vorwärtssprung um *3* Speicherplätze zur Adresse *$400D* erfolgen, um quasi aus der Schleife auszusteigen. Das Verlassen des Programms erfolgt dann über den *RTS*-Befehl.

Schritt 6 (ab Speicherstelle $400A):

Über den neuen *JMP*-Befehl, der übersetzt ***JuMP*** (springe) lautet, wird eine unbedingte Verzweigung, also ein unbedingter Sprung zur angegebenen Adresse definiert. Über diesen *JMP*-Befehl wird die Schleife etabliert, was im Normalfall eine Endlosschleife bedeutet, wäre da nicht die Abbruchbedingung durch den *BEQ*-Befehl, der einen Ausstieg aus dieser Schleife ermöglicht.

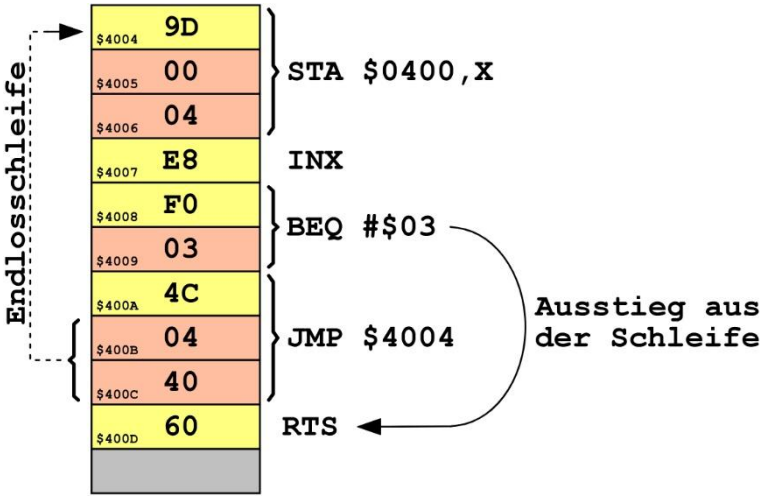


Abbildung 132 - Der Ausstieg aus der Endlosschleife

Schritt 7 (ab Speicherstelle $400D):

Über den *RTS*-Befehl wird das Unterprogramm verlassen, wenn die Schleife verlassen wurde. Sehen wir uns abschließend dazu die Ausführung des Programms an.

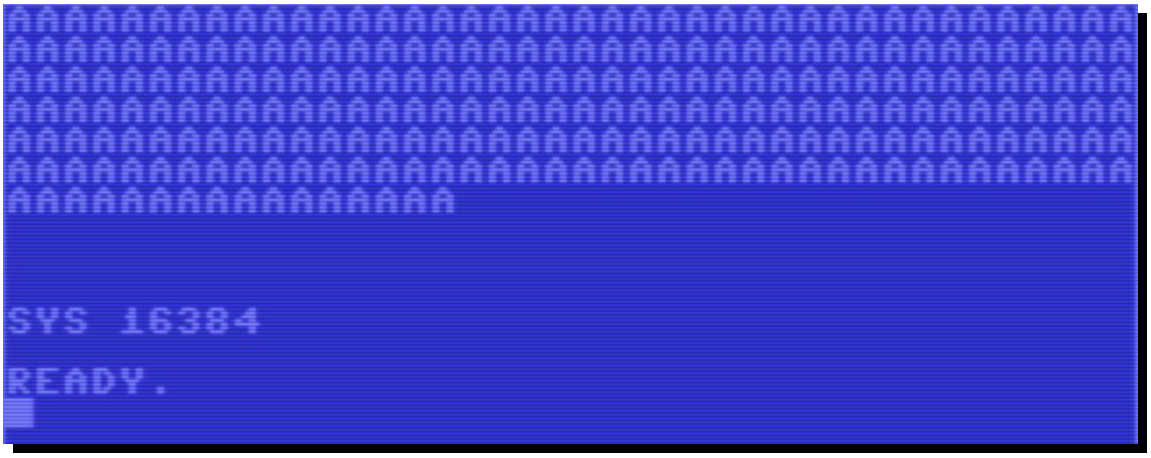


Abbildung 133 - Die Ausführung des Schleifenprogramms

Zusammenfassung der bisherigen Branch-Befehle BNE / BEQ

Ich denke, ich sollte an dieser Stelle noch eine kurze Zusammenfassung der bisher genutzten Branch-Befehle *BNE* und *BEQ* liefern. Diese beiden Befehle nutzen das *Z*-Flag, welches anzeigt, ob die zuletzt durchgeführte Operation eine *0* als Ergebnis geliefert hat. Wenn das der Fall war, dann ist das *Z*-Flag gesetzt und beinhaltet eine *1*.

- ***BNE:*** Hier wird über die relative Adressierung mittels Offset dorthin gesprungen, wenn das *Z*-Flag eine *0* enthält.
- ***BEQ:*** Hier wird über die relative Adressierung mittels Offset dorthin gesprungen, wenn das *Z*-Flag eine *1* enthält.

Es ist sehr einfach, sich mit der Interpretation dieser Funktionalität durcheinander bringen zu lassen.

Weitere Branch-Befehle

Es gibt noch weitere Branch-Befehle, auf die ich nun etwas eingehen möchte. Die folgenden beiden, *BCC* und *BCS* nutzen für ihre Arbeit das Carry-Flag.

- ***BCC:*** Dieser Befehl leitet sich von ***B****ranch if* ***C****arry* ***C****lear* ab und bedeutet, dass dann verzweigt werden soll, wenn das Carry-Flag gelöscht ist.
- ***BCS:*** Dieser Befehl leitet sich von ***B****ranch if* ***C****arry* ***S****et* ab und bedeutet, dass dann verzweigt werden soll, wenn das Carry-Flag gesetzt ist.

Diese vier genannten Branch-Befehle, *BNE*, *BEQ*, *BCC* und *BCS* sind die am meisten genutzten Verzweigungsanweisungen der *6502-/6510-*CPU. Es gibt noch weitere, doch diese werde ich dann zu gegebener Zeit ansprechen. Hier sind sie allesamt in einer Tabelle zu sehen.

Mnemonic	**Bedeutung**	**Erklärung**
BNE	**B**ranch on **N**ot **E**qual	verzweige, wenn Ergebnis der letzten Operation ungleich 0 war (*Z-Flag = 0*)
BEQ	**B**ranch on **EQ**ual	verzweige, wenn Ergebnis der letzten Operation gleich 0 war (*Z-Flag = 1*)
BCC	**B**ranch on **C**arry **C**lear	verzweige, wenn Carry gelöscht (*C-Flag = 0*)
BCS	**B**ranch on **C**arry **S**et	verzweige, wenn Carry gesetzt (*C-Flag = 1*)
BPL	**B**ranch on **PL**us	verzweige, wenn Ergebnis positiv (*N-Flag = 0*)
BMI	**B**ranch on **MI**nus	verzweige, wenn Ergebnis negativ (*N-Flag = 1*)
BVC	**B**ranch on **V C**lear	verzweige, wenn Overflow gelöscht (*V-Flag = 0*)
BVS	**B**ranch on **V S**et	verzweige, wenn Overflow gesetzt (*V-Flag = 1*)

Tabelle 21 - Die Liste der Branch-Befehle

Ein Assemblerprogramm extern schreiben

An dieser Stelle wird es sicherlich interessant, wenn es darum geht, ein längeres Assemblerprogramm zu schreiben und zu editieren. Ich möchte zudem noch mal darauf hinweisen, dass das alles sicherlich viel schneller und eleganter mit der C64-

Studio-Entwicklungsumgebung geht, doch es ist definitiv Geschmackssache und es muss jeder für sich selbst entscheiden, wie er vorgeht. Für mich ist die Sache mit SMON irgendwie etwas authentischer. Es ist auf jeden Fall in SMON recht mühsam und zeitaufwendig, den eingetippten Code an bestimmten Stellen zu erweitern oder ein paar Zeilen einzufügen, ohne, dass bestehender Code überschrieben wird. Also ist es ratsam, das Assemblerprogramm zum Beispiel unter Windows in dem Programm Notepad++ zu schreiben und dann in VICE über *Einfügen* (Paste) den Code zu übernehmen.

Abbildung 134 - Den Code aus der Zwischenablage in VICE einfügen

Es ist dabei wichtig, dass nur Kleinbuchstaben verwendet werden.

```
loop001.asm
 1  a4000
 2  lda #$01
 3  ldy #$00
 4  sta $0400,y
 5  iny
 6  cpy #$08
 7  bne $4004
 8  rts
 9  f
10
```

Abbildung 135 - Das Assemblerprogramm im Notepad++

Damit der letzte Befehl ***F*** für das Abschließen der Assemblereingabe erkannt wird, habe ich einen Zeilenumbruch hinzugefügt, so dass der Cursor in Zeile *10* steht. Es sollte also alles am per ***Strg-A*** markiert und dann über ***Strg-C*** in die Zwischenablage kopiert werden.

Die Addition von zwei 16-Bit Werten

Nun haben wir also die Grundlagen dafür geschaffen, Werte im RAM über die absolute Adressierung abzulegen, dass wir den nächsten Schritt wagen können. Wie schaut es mit der Addition von zwei *16*-Bit Werten aus? Die Vorgehensweise ist analog zur Addition von zwei *8*-Bit Werten, wobei jetzt aber die Zwischenergebnisse im RAM abgelegt werden müssen. Wir erinnern uns, dass der Akku lediglich *8*-Bit breit ist und keine *16*-Bit Werte aufnehmen kann. Wie geht das ganze aber vonstatten?

Die *16*-Bit-Basisarithmetik ist sehr einfach, weil es möglich ist, mithilfe des Carry-Flags *8*-Bit-Operationen einfach

aneinander zu reihen. Wir beginnen einfach mit dem niederwertigsten Byte (LSB) und arbeiten uns bis sich bis zum höchstwertigsten Byte (MSB) vor. Es sollte hier keine Verwirrung hinsichtlich der Potenzen der MSBs aufkommen, denn hinsichtlich einer *16*-Bit Zahl werden die Potenzen von 2^0 bis 2^{15} von rechts nach links aufgeführt. Betrachtet man jedoch LSB und MSB einzeln, sind das dann die Potenzen von je 2^0 bis 2^7. Wir erinnern uns, wie man den Wertebereich über die Angabe der zur Verfügung stehenden Bits ermitteln kann. Bei *16*-Bits schaut das wie folgt aus.

$$\boldsymbol{Anzahl\ Kombinationen} = 2^{16} = 65536$$

Es können also Werte zwischen *0* und *65535* in *16*-Bits gespeichert werden. Das ist doch schon einmal eine ganz gute Steigerung. Zur Realisierung werden im RAM sechs Speicherstellen für die Werte benötigt. Ich nehme dazu einfach die sechs Speicherstellen ab Adresse *$4100*.

Adresse (hex)	Adresse (dez)	Aufgabe
$4100	16640	LO1: LSB Wert 1
$4101	16641	HI1: MSB Wert 1
$4102	16642	LO2: LSB Wert 2
$4103	16643	HI2: MSB Wert 2
$4104	16644	RLO: LSB Resultat-Wert
$4105	16645	RHI: MSB Resultat-Wert

Tabelle 22 - Die Werte im Speicher

Ich möchte als Addition von *16*-Bit-Werten die beiden Dezimalzahlen *11116* und *29121* verwenden. Nun kann ich diese großen Werte nicht in einem *8*-Bit-System speichern. Sehen wir uns dazu die *16*-Bit-Darstellung an.

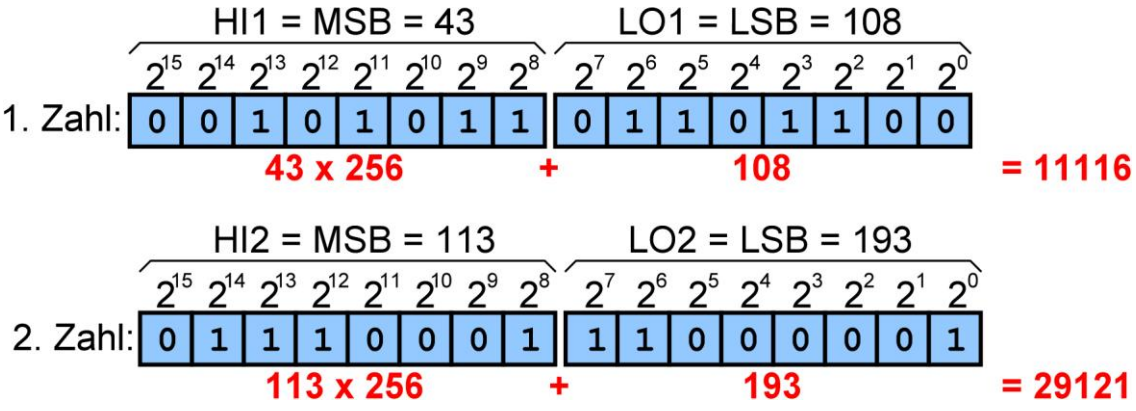


Abbildung 136 - Zwei 16-Bit-Werte

Der oberste Wert *11116* besitzt die gezeigte Bitkombination, die in zwei Hälften von *8*-Bit Werten aufgeteilt ist. Das niederwertige LSB mit *8*-Bits (Dezimalwert = *43*) auf der rechten Seite und das höherwertige MSB mit *8*-Bits (Dezimalwert *108*) auf der linken Seite. Um den daraus resultierenden Wert anhand der zwei *8*-Bit-Werte von LSB und MSB zu berechnen, muss das MSB mit

256 multipliziert und das LSB dazu addiert werden. Gleichermaßen wird mit dem zweiten unteren Wert verfahren. Um die Berechnung mit der *6502-/6510*-CPU durchführen zu können, haben wir jetzt alle erforderlichen Einzel-Bytes (LSB und MSB) der beiden Summanden. Das wären.

- **LO1:** LSB Wert 1 = 108_{10} = *$6C* = 01101100_2
- **HI1:** MSB Wert 1 = 43_{10} = *$2B* = 00101011_2
- **LO2:** LSB Wert 2 = 193_{10} = *$C1* = 11000001_2
- **HI2:** MSB Wert 2 = 113_{10} = *$71* = 01110001_2

Wie schaut denn die Addition dieser beiden Werte quasi zu Fuß aus.

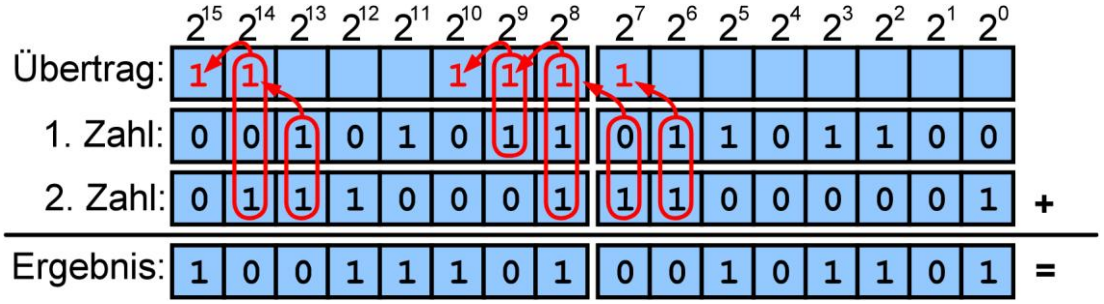


Abbildung 137 - Die Addition zweier 16-Bit-Werte

Das Ergebnis lautet in dezimaler Schreibweise *40237*. Sehen wir uns also das entsprechende Maschinenprogramm an. Die Vorgehensweise ist wie folgt.

- Einmaliges Löschen des Carry-Flags
- Addition der *LOW*-Bytes beider Werte
- Addition beider *HIGH*-Bytes beider Werte unter der Berücksichtigung des Carry-Flags

Nachfolgen habe ich noch einmal eine grafische Übersicht über die Adressen der einzelnen Werte erstellt.

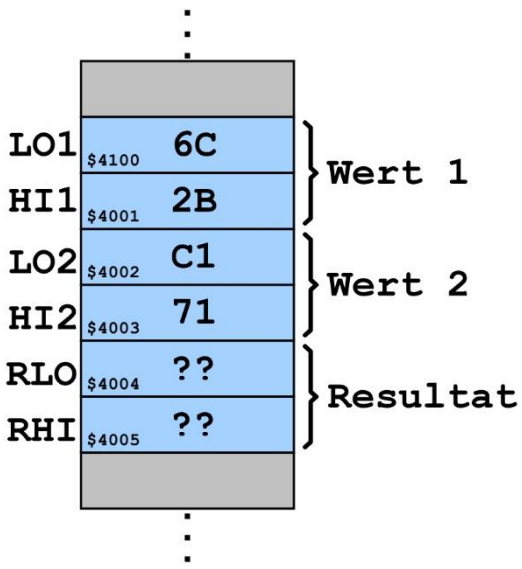


Abbildung 138 - Die Werte im Speicher

Doch nun zum Programm.

```
.4000   A9 6C      LDA #6C
.4002   8D 00 41   STA 4100
.4005   A9 2B      LDA #2B
.4007   8D 01 41   STA 4101
.400A   A9 C1      LDA #C1
.400C   8D 02 41   STA 4102
.400F   A9 71      LDA #71
.4011   8D 03 41   STA 4103
.4014   18         CLC
.4015   AD 00 41   LDA 4100
.4018   6D 02 41   ADC 4102
.401B   8D 04 41   STA 4104
.401E   AD 01 41   LDA 4101
.4021   6D 03 41   ADC 4103
.4024   8D 05 41   STA 4105
.4027   00         BRK
```

Abbildung 139 - Die Addition zweier 16-Bit-Werte

Von der Adresse *$4000* bis *$4013* werden die Werte der Summanden zuerst in den Akku geladen und dann an die betreffenden Speicherstellen geschrieben.

Ab Speicherstelle *$4014* beginnt der eigentliche Rechenvorgang, wobei nur einmalig das Carry-Flag über den *CLC*-Befehl gelöscht werden darf. Es werden die Daten der zuvor abgelegten Werte aus den betreffenden Speicherstellen gelesen und über die zwei Additionen in die Speicherstellen für das Resultat geschrieben.

Starten wir das Programm über

G4000

und sehen dann nach, wie die Speicherinhalte für die Daten aussehen.

Abbildung 140 - Die Ausführung des Programms und die Anzeige der Daten

Es ist zu sehen, dass hier ein neuer SMON-Befehl angewendet wurde. Über die Eingabe vom

M4100

können die ersten acht Bytes mit deren Inhalten angezeigt werden. Bei jedem weiteren Druck auf die Leertaste kommt eine weitere Zeile der nachfolgenden Adressen hinzu. Abgebrochen wird dieser Vorgang in VICE mit einem Druck auf die *ESC*-Taste.

Es ist zu sehen, dass sich an den ersten vier Speicherstellen ab Adresse *$4100* genau die Werte befinden, die durch das Programm vorgegeben wurden. Das Ergebnis befindet sich nach der Programmausführung in den Speicheradresse *$4104* (RLO) beziehungsweise *$4105* (RHI). Das sind in unserem Fall.

- **$4104: $2D (dezimal 45)**
- **$4105: $9D (dezimal 157)**

Das Ergebnis der *16*-Bit-Addition errechnet sich nun wie folgt.

$$\boldsymbol{Ergebnis} = RLO + 256 \cdot RHI = 45 + 256 \cdot 157 = 40237$$

Das passt also. Man kann das Ergebnis auch über bestimmte Befehle in Basic ermitteln.

Abbildung 141 - Das Ergebnis über BASIC

Wenn man in BASIC des Öfteren mit Low- und High-Bytes arbeitet, kann man sich eine Funktion zur Umrechnung und Anzeige schreiben. Über die Eingabe des Wertes für das Low-Byte wird die darauffolgende Speicherstelle für das High-Byte implizit verwendet.

```
10 DEF FNADR(X)=PEEK(X)+256*PEEK(X+1)
20 INPUT"LOW-BYTE: ";X
30 PRINT"ADRESSE= ";FNADR(X)
READY.
```

Abbildung 142 - Das BASIC-Programm zur Berechnung der Speicheradresse

Geben wir einen Wert für das Low-Byte ein.

```
RUN
LOW-BYTE:  ? 45
ADRESSE=   2126

READY.
```

Abbildung 143 - Das BASIC-Programm berechnet die Speicheradresse

Die Subtraktion

Kommen wir nun zur gegenteiligen Operation der Addition: Die *Subtraktion*. Um eine Binärzahl von der anderen quasi abzuziehen, muss einiges beachtet werden. Ich bin ja schon auf das Einer- bzw. das Zweierkomplement eingegangen und somit sollte ein gewisses Verständnis für positive und negative Werte und deren Darstellung vorhanden sein. Beginnen wir mit einem einfachen Beispiel, was auf den ersten Blick nun überhaupt nichts mit der Subtraktion zu tun hat. Ich möchte zwei verschiedene Binärzahlen in den Akku laden und dann gleichzeitig das Status-Register des jeweiligen Wertes untersuchen. Nichts Spektakuläres also. Die beiden Werte schauen wir folgt aus.

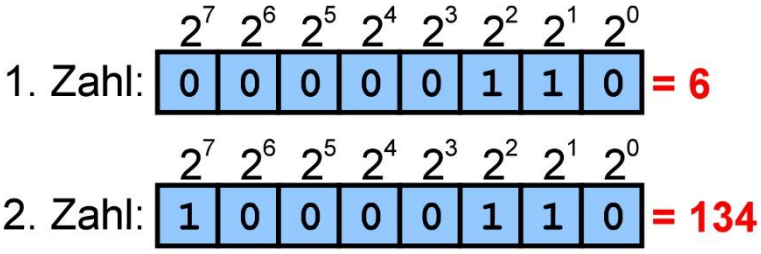


Abbildung 144 - Zwei ganz normale Werte

Die beiden dezimalen Werte schauen hinsichtlich ihrer Bitkombination fast gleich aus und lediglich beim untern ist das *MSB* (Most-Significant-Bit) gesetzt. Ich möchte an einem kleinen Programm zeigen, wie sich das Laden dieser Werte in den Akku unmittelbar auf das Statusregister respektive das *N*-Flag auswirkt. Das folgende Programm löscht zuerst das Carry-Flag über den *CLC*-Befehl (ist eigentlich hier nicht von Bedeutung) und lädt im Anschluss nacheinander unterschiedliche Werte über den *LDA*-Befehl in den Akku. Wir wollen die Auswirkungen auf das *N*-Flag untersuchen.

```
.D4000
,4000  18        CLC
,4001  A9 00     LDA #00
,4003  A9 06     LDA #06
,4005  A9 86     LDA #86
,4007  00        BRK
--------------------------
.
```

Abbildung 145 - Das Laden von Werten in den Akku

In SMON gibt es einen Befehl, der das im Speicher vorliegende Programm schrittweise abarbeiten kann. Dieses *Tracen*, was übersetzt Ablaufverfolgung bedeutet, wird in der Programmierung zur Analyse von Programmen und zur Fehlersuche - auch *Debugging* genannt - genutzt. Der Aufruf erfolgt allgemein über die Eingabe von

TW <Startadresse>

Das Kürzel *TW* steht hier für *Trace-Walk*, also einen schrittweise ausgeführten Programmablauf ab der genannten Speicheradresse. SMOM stoppt nach der Ausführung und zeigt dann die Inhalte aller Register in der gleichen Reihenfolge wie bei der Eingabe des *R*-Kommandos an. Wir rufen uns das noch mal ins Gedächtnis.

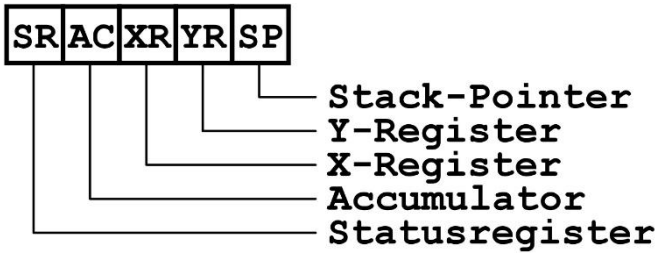


Abbildung 146 - Die Registeranzeige des R-Kommandos

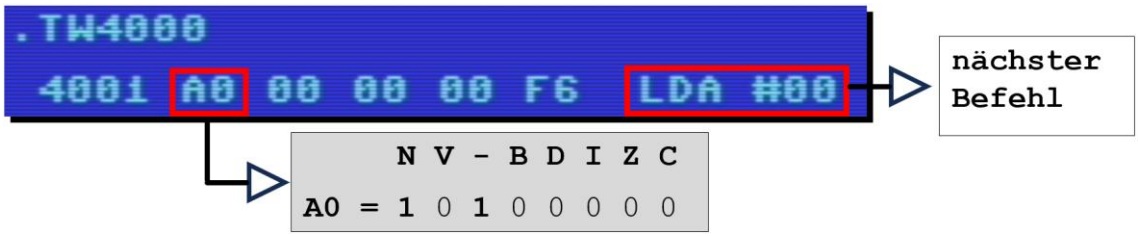
Leider wird das für uns so interessante Statusregister nicht wie beim R-Kommando in einzelne Bits aufgeschlüsselt, so dass dann eine einfachere Interpretation derselben möglich wäre. Wir müssen uns den angezeigten hexadezimalen Wert in einen Binärwert umrechnen. Fangen wir an. Es ist wichtig zu erkennen, dass beim Aufruf und dem schrittweise Abarbeiten der Zeilen immer die aktuellen Registerinhalte und der nächste auszuführende Befehl angezeigt werden. Dieser Befehl, wird also erst beim Druck auf die Leertaste ausgeführt, wobei wiederum der nächste auszuführende Befehl in Erscheinung tritt! Fangen wir an.

TW4000

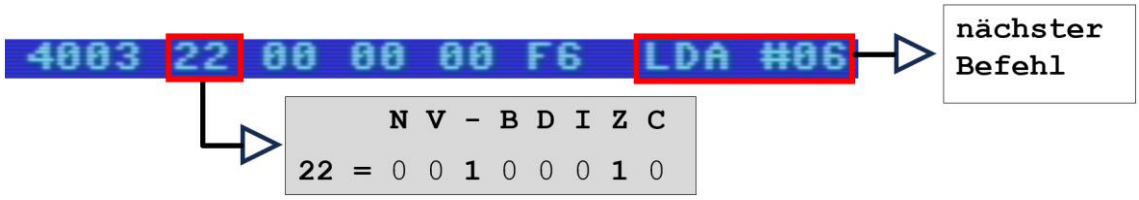
Der erste Befehl *CLC* an Speicherstelle *$4000* wurde hierbei schon ausgeführt!

```
.TW4000
 4001 A0 00 00 00 F6  LDA #00
```

Wie ist das hier zu interpretieren? Es ist zu sehen, dass das Statusregister *SR* mit dem Wert *$A0* geladen ist und die Flags den gezeigten Status besitzen. Vom vorherigen *CLC*-Befehl ist zu erkennen, dass das *Carry*-Flag gelöscht ist. Alles andere ist erst einmal zweitranging. Der nächste auszuführende Befehl *LDA #$00* ist zu sehen.

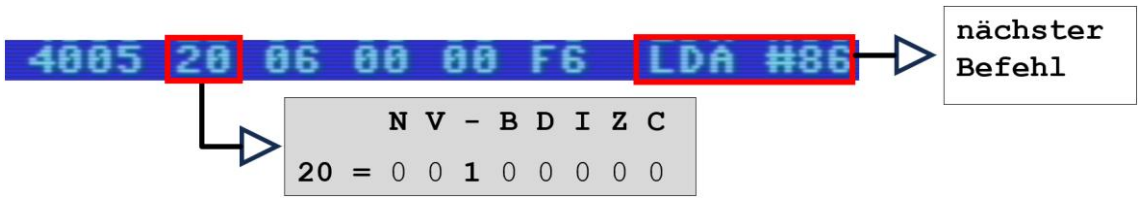


Drücken wir jetzt 1x die Leertaste.



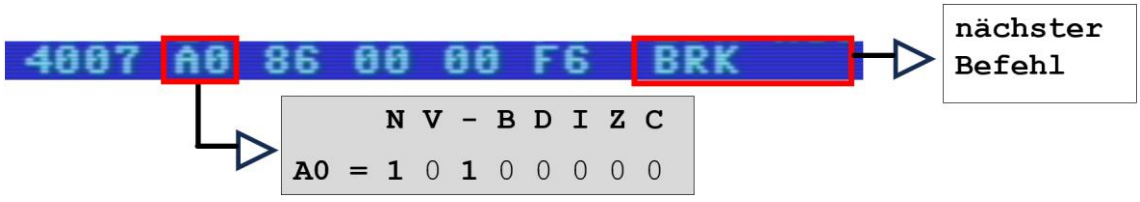
Der Akku wurde mit dem Wert *#$00* und das Statusregister *SR* mit dem Wert *22* geladen, was bedeutet, dass das *Z*-Flag (Null-Wert im Akku) gesetzt wurde. Zudem ist zu sehen, dass das *N*-Flag nun den Wert *0* besitzt, weil die Interpretation des Wertes im Akku als positiv angesehen wird. Also Bit *7* ist nicht gesetzt, was für alle Werte kleiner *128* zutrifft. Der nächste auszuführende Befehl lautet *LDA #$06*.

Drücken wir jetzt 1x die Leertaste.



Der Akku wurde mit dem Wert *#$06* und das Statusregister *SR* mit dem Wert *20* geladen, was bedeutet, dass das *Z*-Flag wieder auf *0* gesetzt wurde, weil im Akku kein Null-Wert mehr vorhanden ist. Das *N*-Flag bleibt weiterhin gelöscht, weil es sich um einen positiven Wert im Akku handelt. Der nächste auszuführende Befehl lautet *LDA #$86*.

Drücken wir jetzt 1x die Leertaste.



Der Akku wurde mit dem Wert *$86* und das Statusregister *SR* mit dem Wert *A0* geladen, was bedeutet, dass das *Z*-Flag immer noch auf *0* gesetzt ist, weil im Akku kein Null-Wert mehr vorhanden ist. Doch diesmal wurde das *N*-Flag gesetzt, da der Wert *$86* als negativ interpretiert wird. Der nächste auszuführende Befehl lautet *BRK* und würde den Programmlauf im Normalfall unterbrechen.

Nun wollen wir diese Rechnung einmal mit der CPU umsetzen. Dazu verwenden wir folgende neue OP-Codes. Es handelt sich um *SBC* (Subtract with Carry) und *SEC* (Set Carry). Zum einen benötigen wir natürlich einen Befehl, der zwei Werte subtrahiert und dafür wird *SBC* verwendet, der ähnlich wie der schon bekannte *ADC* zur Addition arbeitet.

E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX i	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ r	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED i	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 147 - Die OP-Code-Matrixtabelle für den Op-Code SBC

Das **#** unterhalb des *SBC*-Befehls bedeutet, dass es sich um eine unmittelbare Adressierung handelt.

N	V	-	B	D	I	Z	C
✓	✓	-	-	-	-	✓	✓

Tabelle 23 - Die Flag-Beeinflussung des SBC-Befehls

Operation: A - M - C → A (Subtraktion von Akku, Wert und gesetztem Carry. Speicherung erfolgt im Akku)

Und zum anderen haben wir den *SEC*, der das Carry-Flag setzt, weil die CPU intern mit dem erwähnten Zweierkomplement arbeitet und aus diesem Grund dieses gesetzt werden muss.

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3

Abbildung 148 - Die OP-Code-Matrixtabelle für den Op-Code SEC

Das ***I*** unterhalb des *SEC*-Befehls bedeutet, dass es sich um eine implizite Adressierung handelt.

N	V	-	B	D	I	Z	C
-	-	-	-	-	-	-	1

Tabelle 24 - Die Flag-Beeinflussung des SEC-Befehls

Operation: 1 → C (Das Carry-Flag wird gesetzt)

Sehen wir uns dazu nun einen Assembler-Code an, der die Differenz von *56* und *3* bildet.

```
,4000  38        SEC
,4001  A9 38     LDA #38
,4003  E9 03     SBC #03
,4005  00        BRK
--------------------------
.
```

Abbildung 149 - Eine einfache Subtraktion

Zu Beginn wird über den *SEC*-Befehl das Carry-Flag gesetzt. Im Anschluss wird der dezimale Wert *56* (*#$38*) in den Akku geladen und mit dem darauffolgenden SBC-Befehl die Subtraktion mit dem Wert *3* vorgenommen. Starten wir das Programm wieder mit

G4000

```
  PC  SR AC XR YR SP  NV-BDIZC
;C00B B0 C2 00 00 F6  10110000
.G4000

  PC  SR AC XR YR SP  NV-BDIZC
;4006 31 35 00 00 F6  00110001
.
```

Abbildung 150 - Das Programm wurde gestartet

Im Akku befindet sich das Ergebnis *#$35* dieser Subtraktion. Ich möchte an dieser Stelle die Sache mit dem Carry-Flag einmal etwas beleuchten. Dieses Flag zeigt also an, wenn bei einer arithmetischen Operation ein Übertrag auf die nächst höhere Stelle aufgetreten ist. Also zum Beispiel bei der Addition von *255 + 1*. Das Ergebnis wäre *100000000* und würde *9* Bits zur Speicherung erfordern. Das höchstwertigste Bit (1) besitzt dabei den Stellenwert *256*. Also keine Chance für *8*-Bits! Bei der arithmetischen Operation *ADC* (Add-With-Carry) wird dieser Übertrag also berücksichtigt und das Carry-Flag mit hinzuaddiert. Bei der Subtraktion über *SBC* (Subtract-With-Carry) wird das Carry-Flag ebenfalls berücksichtigt, wobei es im Grunde genommen nicht Übertrag, sondern *Borger* genannt wird. Im Mathematikunterricht haben wir gelernt, dass bei einer Subtraktion, bei der der Minuend kleiner als der Subtrahend ist, ein sogenannter Borger erforderlich ist. Das ist am folgenden Beispiel einfach zu erkennen.

```
  125  (Minuend)
-  36  (Subtrahend)
  ---
   89  (Differenz)
```

Wenn man also von rechts nach links mit der Differenzbildung der einzelnen Stellen beginnt, stellt man ganz zu Beginn fest, dass *5 - 6* nicht geht, weil *5* kleiner als *6* ist. Was macht man in diesem Fall also? Ganz einfach! Es muss eine *10* (die Basiszahl bei der Dezimalrechnung) von der nächsten Spalte ausgeliehen werden zum Minuenden hinzugefügt werden. Das würde dann in meinem Fall *15 - 6* ergeben. Diese ausgeliehene - geborgte - *10* wird dann wieder auf den Subtrahenden der nächsten Spalte zurückgesetzt, sobald die Differenz gebildet wurde. Bei der Subtraktion von Binärzahlen kommt das gleiche Verfahren zur Anwendung, wie bei der Subtraktion von Dezimalzahlen, wobei hier natürlich die Basis *2* zur Anwendung kommt. Es gelten folgende Regeln.

```
          Ü
0 - 0 = 0 0
1 - 0 = 1 0
1 - 1 = 0 0
0 - 1 = 1 1
```

Abbildung 151 - Rechenregeln für eine Subtraktion

Die Subtraktion von *0 - 1* führt zu einem negativen Ergebnis, was zu Problemen führt. Aus diesem Grund kommt hier wieder der Übertrag zum Zuge. Die einfachste Vorgehensweise zur Bildung einer Differenz bei Binärzahlen ist dann wirklich die Bildung des Zweierkomplementes des Subtrahenden und die nachfolgende Addition beider Werte.

Wir sollte folgendes für eine Addition und eine Subtraktion auf jeden Fall festhalten.

- **Addition:** Vor einer anstehenden Addition muss das Carry-Flag immer mit dem *CLC*-Befehl gelöscht werden.
- **Subtraktion:** Vor einer anstehenden Subtraktion muss das Carry-Flag immer mit dem *SEC*-Befehl gesetzt werden.

Die Multiplikation

So, ich denke, wir können mit der Multiplikation beziehungsweise Division fortfahren. Die *6502-/6510*-CPU kennt zwar Befehle für die Addition und die Subtraktion, wie wir das gerade eben gesehen haben, doch es gibt nichts in dieser Richtung für die Multiplikation oder Division. Man muss sich also irgendwie zu helfen wissen und eine Addition - damit fangen wir an - als eine

Abfolge von nacheinander ausgeführten Additionen sehen. Wenn man zum Beispiel die folgende einfache Multiplikation hat

6 x 5

kann das natürlich auch wie folgt geschrieben werden, denn der Wert *5* soll ja in der Rechnung sechs Mal vorkommen beziehungsweise in einer entsprechenden Addition untereinandergeschrieben werden. Das wäre dann

5 + 5 + 5 + 5 + 5 + 5

Lassen wir uns dieses einfache Beispiel zuerst einmal in einem Flussdiagramm anschauen. Wir brauchen dazu natürlich den Akkumulator für die eigentliche Rechenoperation und dann eine Art von Zähler, der kontrolliert, wie oft denn eine gewünschte Operation durchzuführen ist. Ich nutze diesmal das *Y*-Register.

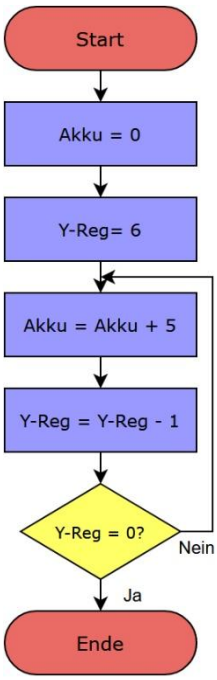


Abbildung 152 - Flussdiagramm für eine einfache Multiplikation

Nun werden wir das Flussdiagramm in ein entsprechendes Assembler-Programm übersetzen, das dann wie folgt ausschaut.

```
,4000   18        CLC
,4001   A9 00     LDA #00
,4003   A0 06     LDY #06
,4005   69 05     ADC #05
,4007   88        DEY
,4008   D0 FB     BNE 4005
,400A   00        BRK
--------------------------------
.
```

Abbildung 153 - Die wiederholte Addition

Bei der Multiplikation *6 x 5* kommt ja bekannterweise *30* heraus. Sehen wir nach und starten das Programm mit

G4000

```
.G4000
  PC  SR AC XR YR SP  NV-BDIZC
;400B 32 1E 06 00 E6  00110010
.
```

Abbildung 154 - Das Programm wurde gestartet

Es ist zu sehen, dass der Inhalt des Akkus *1E* beträgt und das ist in dezimaler Schreibweise eben *30*. Natürlich darf das Ergebnis in dieser Weise nicht größer dem Wert *255* sein, da das Carry-Bit dann nicht berücksichtigt wird. Sicherlich ist aber das auch möglich und wir werden uns das anhand des folgenden Beispiels ansehen. Dabei wird - wie zuvor - eine Zahl in einer Schleife mehrfach addiert. Tritt diesmal je Schleifendurchlauf ein Übertrag auf, wird er in einer Speicherstelle (hier *$4100*) quasi gesammelt und aufaddiert. Die folgende Grafik wird das sicherlich verdeutlichen.

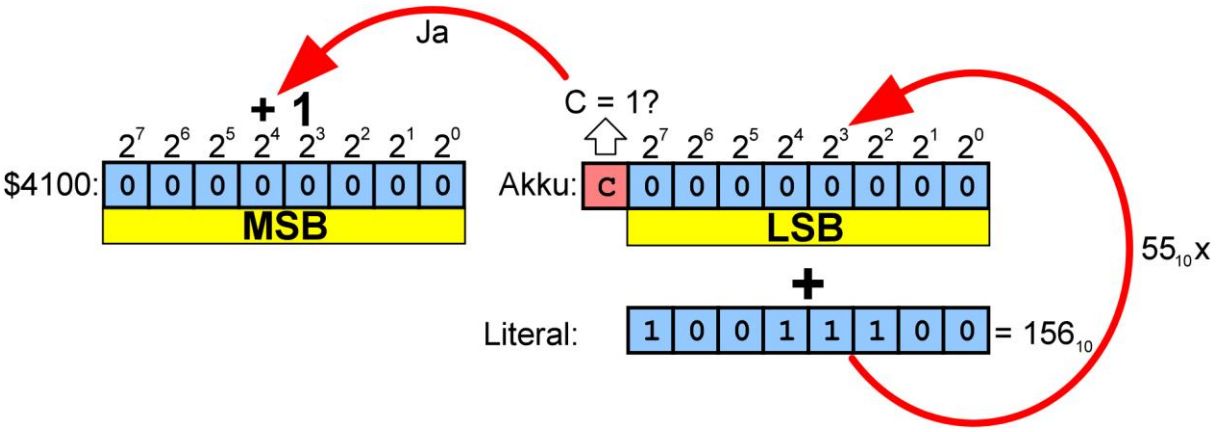


Abbildung 155 - Die Multiplikation von zwei 8-Bit-Werten mit Überlauf

Doch sehen wir uns vor dem Assembler-Programm das entsprechende Flussdiagramm an. Die Funktion hinsichtlich des möglichen Übertrags ist recht einfach. Immer, wenn über den *BCC*-Befehl kein Übertrag aus der letzten Operation festgestellt wurde, kommt es nicht zur Inkrementierung der Speicherstelle *$4101*.

Diese Speicherstelle protokolliert die aufgetretenen Überträge.

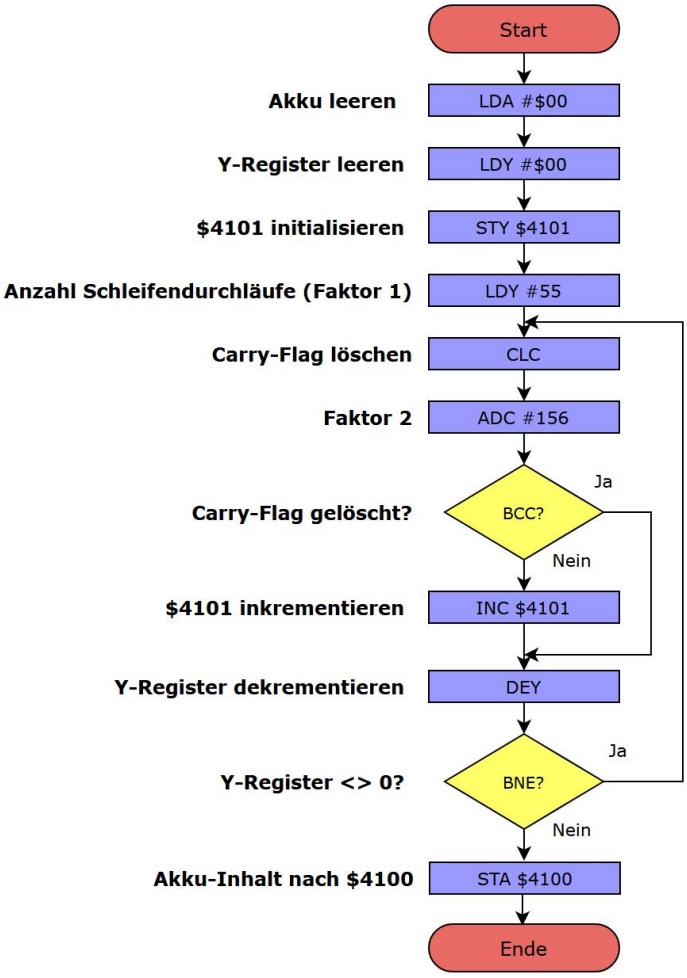


Abbildung 156 - Flussdiagramm Multiplikation zweier großer 8-Bit-Zahlen

Kommen wir nun zum eigentlichen Assembler-Programm. Ich habe mit für die Multiplikation der folgenden Werte entschieden.

55 x 156 = ???

Die hexadezimalen Werte lauten:

55_{10} = $37

156_{10} = $9C

Nach der Abarbeitung des Programms sind die folgenden beiden Speicherstellen mit dem Ergebnis der durchgeführten Multiplikation befüllt.

- **$4100: LSB** (aus dem Akku)
- **$4101: MSB** (aus den Überträgen)

Das Programm gestaltet sich wie nachfolgend zu sehen ist. Ich habe hier schon die entsprechenden bedingten Sprünge eingezeichnet.

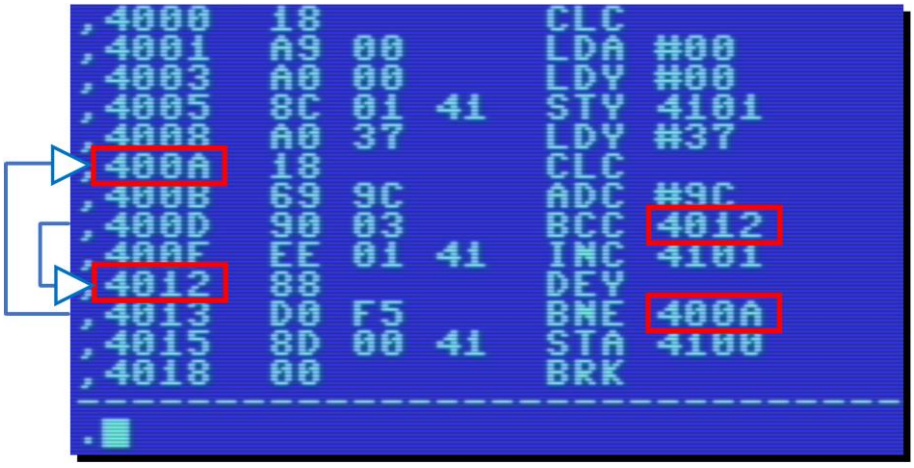
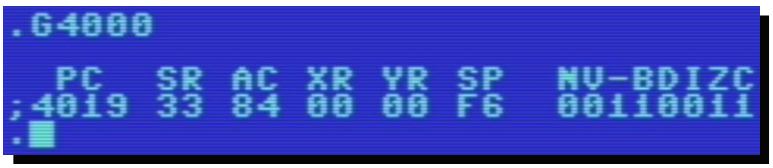


Abbildung 157 - Die Multiplikation mit Überlauf

Führen wir das Programm wieder mit der Eingabe von

G4000

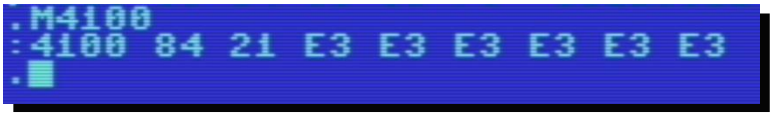
aus.



Hier ist schon einmal das Ergebnis für das LSB ($84) zu sehen, das sich ja im Akku befindet. Dieser Wert wurde durch das Programm in Speicherstelle *$4100* kopiert. Der Wert für das MSB befindet sich in Speicherstelle *$4101*. Wir können diese beiden Inhalte durch die folgende Eingabe

M4100

sichtbar machen.



Der Inhalt der Speicherstelle *$4101* beträgt also *$21*. Rechnen wir das Ergebnis nach der bekannten Vorgehensweise aus.

- $84 = 132_{10}
- $21 = 33_{10}

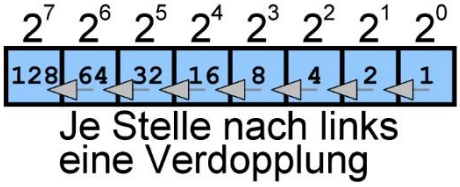
$$\boldsymbol{Ergebnis} = LSB + 256 \cdot MSB = 132 + 256 \cdot 33 = 8580$$

Es stimmt also...

55 x 156 = 8580

Verblüffend, nicht wahr?! Eine Multiplikation auf diese Weise führt natürlich dazu, dass bei großen Werten, die Schleifen recht oft durchlaufen werden müssen. Es sollte dann derart realisiert

werden, dass der kleinere Summand zur Bildung des Produktes für die Schleifendurchläufe verwendet wird. Hinsichtlich der der Stellenwertigkeiten bei Binärzahlen ist etwas Interessantes zu bemerken. Das *LSB*, also das niederwertige Bit besitzt ja die Wertigkeit *1* und wenn man je eine Stelle nach links zum MSB wandert, kommt es zu einer Verdopplung des Wertes.



Was mit einem einzelnen Bit funktioniert, geht natürlich auch mit mehreren Bits.

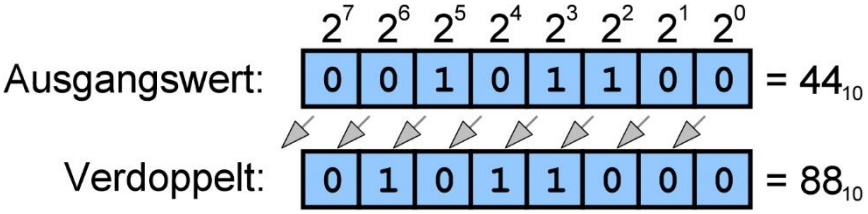


Abbildung 158 - Der Ausgangswert wurde verdoppelt

Natürlich ist auch hier wieder darauf zu achten, ob zum Beispiel das Carry-Bit bei einer entsprechenden Rotation oder Verschiebung gesetzt wurde. Darauf gehe ich aber nun nicht weiter ein. Die Schieberei wird in einem gesonderten Kapitel behandelt. Ein bisschen Geduld also noch!

Die Division

Wie schon erwähnt, besitzt die *6502-/6510*-CPU von Hause aus auch keine Möglichkeit, eine Division durchzuführen. Analog zur Multiplikation kann eine Division als eine wiederholte Subtraktion gesehen werden. Ich möchte das an dem folgenden kleinen Beispiel verdeutlichen und die Sache nicht mit größeren Werten vertiefen. Angenommen, man möchte den Wert *40* durch *2* dividieren. Hierbei wird der Akku dafür verwendet, den entstandenen Rest der Division aufzunehmen. Also beginnend mit *40*, *38*, *36*, *34* bis zu ..., *6*, *4*, *2*, *0*. Ich verwende sowohl *X*- als auch *Y*-Register, wobei das *X*-Register dazu verwendet wird, den Divisor im Speicher abzulegen und das *Y*-Register, die Anzahl der durchgeführten Subtraktionen. Am Ende des Programms enthält also das *Y*-Register den Quotienten. Hier noch einmal zur Verdeutlichung der verwendeten Begriffe.

$$Quotient = \frac{Dividend}{Divisor}$$

Sehen wir uns dazu das folgende Programm an, wo *148* (*$94*) durch *2* (*$02*) gerechnet wird.

```
.4000   A0 00      LDY #00
.4002   A2 02      LDX #02
.4004   8E 00 41   STX 4100
.4007   A9 94      LDA #94
.4009   38         SEC
.400A   E9 02      SBC #02
.400C   C8         INY
.400D   CD 00 41   CMP 4100
.4010   B0 F7      BCS 4009
.4012   00         BRK
--------------------------
.
```

Abbildung 159 - Die Division

Nach dem Start des Programms über

G4000

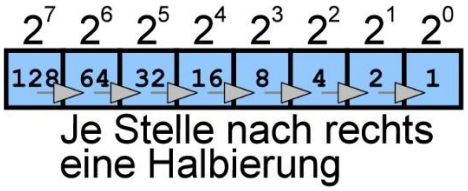
ist die folgende Anzeige zu sehen, wobei unser Augenmerk auf dem Inhalt des *Y*-Registers liegen sollte. Dort ist das Ergebnis gespeichert.

```
.G4000
  PC  SR AC XR YR SP  NV-BDIZC
;4013 B0 00 02 4A F6  10110000
.
```

Abbildung 160 - Das Ergebnis der Division liegt im Y-Register

Im Y-Register ist der Wert *$4A* zu sehen, was dem dezimalen Wert *74* entspricht.

Was eben bei der Multiplikation hinsichtlich der der Stellenwertigkeiten bei Binärzahlen zu bemerken war, funktioniert natürlich auch in die andere Richtung. Wenn man eine Bitkombination je eine Stelle nach rechts zum *LSB* hin verschiebt, kommt es zu einer Halbierung des Wertes.



Natürlich geht das auch in diesem Fall mit mehreren Bits.

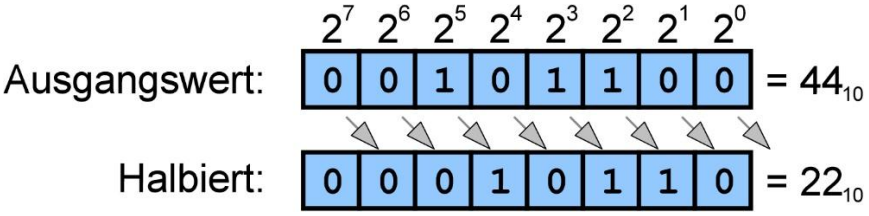


Abbildung 161 - Der Ausgangswert wurde halbiert

Auch hier wieder darauf zu achten, ob zum Beispiel das Carry-Bit bei einer entsprechenden Rotation oder Verschiebung gesetzt wurde.

Wir vergleichen

Wir vergleichen

Wenn ein Programmablauf nicht geradlinig verlaufen soll, sondern in Abhängigkeit bestimmter Umstände respektive Werte, dann müssen Vergleiche durchgeführt werden. Es gibt hier die drei Vergleichsbefehle, wobei ich den *CPY*-Befehl für einen Zähler schon genannt hatte.

- CMP
- CPX
- CPY

Vergleichen heißt im Englischen *Compare*, und daher beginnen diese Befehle mit einem ***C***. Für diese Befehle können zum einen die unmittelbare Adressierung genutzt werden, was bedeutet, dass der Akku mit einem festen Wert verglichen werden soll.

CMP #$17

Der Wert *$17* wird mit dem Inhalt des Akkus verglichen. Zum anderen kann natürlich auch die absolute Adressierung Verwendung finden, was bedeutet, dass der Akku mit einem Wert der angegebenen Speicherstelle verglichen werden soll.

CPX $4100

Der Wert, der sich in der Speicherstelle *$4100* befindet, soll mit dem Akku verglichen werden.

Finden Vergleiche zwischen Akku und einem Wert statt, kommen als Ergebnis nur drei Antworten in Frage.

- Der Akku-Inhalt war größer (>)
- Der Akku-Inhalt war gleich (=)
- Der Akku-Inhalt war kleiner (<)

Nun sind das drei Möglichkeiten, die mit der Abfrage eines einzelnen Flags des Statusregisters nicht zu realisieren ist. Genutzt werden hier die folgenden Flags.

- N-Flag
- Z-Flag
- C-Flag

In der folgenden Darstellung ist das Verhalten der Flags in Abhängigkeit des Vergleichs zu sehen.

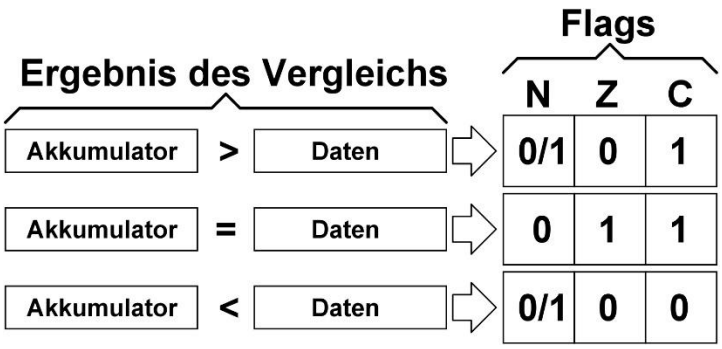


Abbildung 162 - Die Flag-Beeinflussung bei Vergleichsoperationen

Nachfolgend habe ich das mit ein paar unterschiedlichen Werten durchgespielt.

Der Akku-Inhalt ist größer:

```
.D4000
,4000  A9 F5     LDA #F5
,4002  C9 A4     CMP #A4
,4004  00        BRK
-----------------------------
.G4000

  PC  SR AC XR YR SP  NV-BDIZC
;4005 31 F5 00 00 F7  00110001
.
```

Akku > Daten

Der Akku-Inhalt ist gleich:

```
.D4000
,4000  A9 C4     LDA #C4
,4002  C9 C4     CMP #C4
,4004  00        BRK
-----------------------------
.G4000

  PC  SR AC XR YR SP  NV-BDIZC
;4005 33 C4 00 00 F7  00110011
.
```

Akku = Daten

Der Akku-Inhalt ist kleiner:

```
.D4000
,4000  A9 17     LDA #17
,4002  C9 B9     CMP #B9
,4004  00        BRK
-----------------------------
.G4000

  PC  SR AC XR YR SP  NV-BDIZC
;4005 30 17 00 00 F7  00110000
.
```

Akku < Daten

Bit-Spielereien

- Bits setzen
- Bits löschen
- Bits toggeln - invertieren
- Bits schieben

Bit-Spielereien

Das nun folgende Thema soll eine kleine Einführung in die Manipulation von Bits und Bytes sein und wiederum kann ich nur an der Oberfläche kratzen. Ich versuche aber die wichtigsten Bereiche einmal kurz zu streifen. Im Grunde genommen kann man die Bit-Manipulationen in die folgenden Punkte aufteilen.

- Ein Bit setzen
- Ein Bit löschen
- Ein Bit toggeln (logisch umkehren)
- Ein Bit abfragen
- Bits verschieben oder rotieren

Bits setzen

Wenn es darum geht, Einfluss auf ein bestimmtes Bit zu nehmen und es zum Beispiel zu setzen - es besteht auch die Möglichkeit, dass es vorher schon gesetzt war - bedeutet dies, dass alle anderen Bits unangetastet bleiben und ihren bisherigen Pegel beibehalten. Angenommen, die folgende Bit-Kombination eines Bytes ist gegeben.

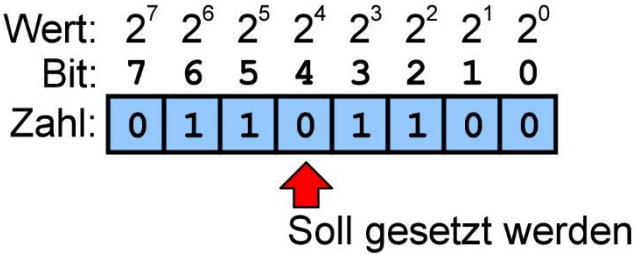


Abbildung 163 - Ausgangs-Bit-Kombination für das Setzen eines Bits

Wie zu sehen ist, möchte ich jetzt das Bit 4 - was noch 0 ist - setzen, ohne, dass andere Bits in irgendeiner Weise beeinflusst werden. Zu diesem Zweck kommt wieder eine logische Verknüpfung ins Spiel. Welche Verknüpfung kommt dabei in Betracht, die am Ausgang eine *1* liefert, egal, ob der erste Eingang mit einer *0* oder einer *1* versehen ist? Richtig, die *ODER*-Verknüpfung (*OR*-Verknüpfung). Wir sehen uns noch einmal die entsprechende Wertetabelle dazu an.

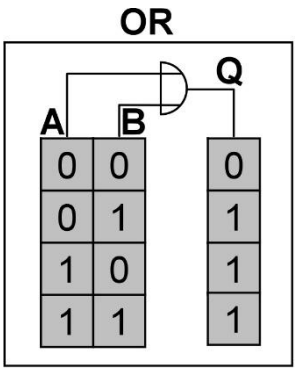


Abbildung 164 - Eine ODER-Verknüpfung

Es ist zu sehen, dass eine *1* an einem Eingang ausreicht, den Ausgang ebenfalls auf *1* zu setzen. Eine Bit-Maske, die das Setzen von Bits bewirkt, schaut wie folgt aus. Alle Bits, die gesetzt werden sollen, müssen dabei in der Bit-Maske mit einer *1* versehen sein, die unverändert bleiben sollen, müssen in der Bit-Maske mit einer 0 gekennzeichnet sein.

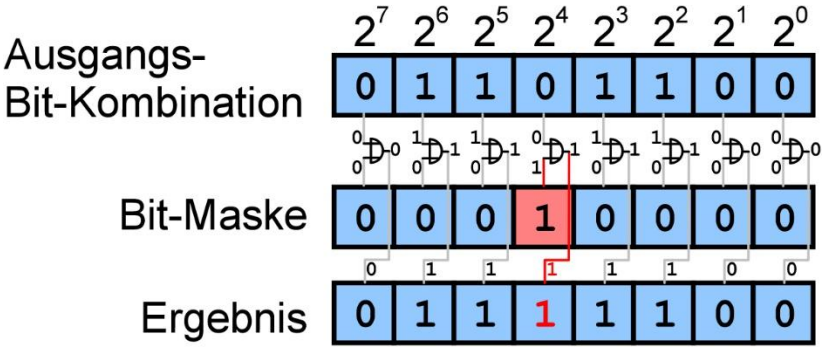


Abbildung 165 - Ein Bit wird gesetzt

Das gewünschte Bit an Position *4* wurde gesetzt und alle anderen haben den gleichen Pegel wie vorher. Natürlich kann auf diese Weise mehr als ein Bit gesetzt werden. Dazu muss lediglich die Bit-Maske an den entsprechenden Stellen mit Einsen versehen sein. Sehen wir uns dazu wieder ein kleines Assembler-Programm mit anderen Werten an. Genutzt wird der *ORA*-Befehl.

Nachfolgend die Matrix für den *ORA*-Befehl.

OP-Code: 09

MSD	W65C02S OpCode Matrix															MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0

Abbildung 166 - Die OP-Code-Matrixtabelle für den Op-Code ORA

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 25 - Die Flag-Beeinflussung des ORA-Befehls

Operation: A V M → A (Der ORA-Befehl überträgt den Speicher/Wert und den Akkumulator an den Addierer, der eine binäre ODER-Verknüpfung durchführt und das Ergebnis im Akkumulator speichert.)

Hier nun das Programm.

```
.D4000
,4000   A9 85      LDA #85
,4002   09 F0      ORA #F0
,4004   00         BRK
--------------------------------
.
```

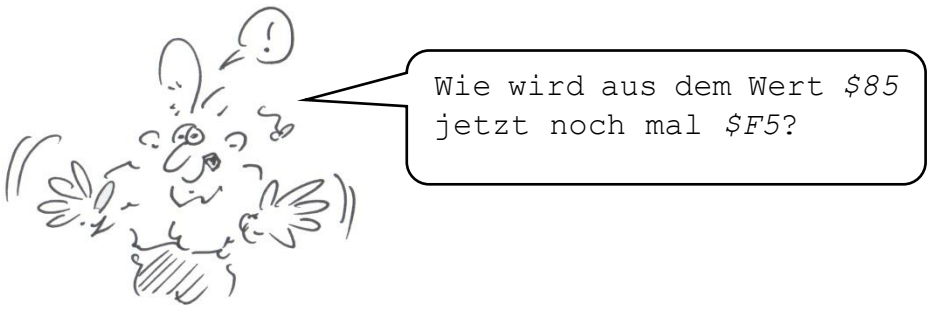
Abbildung 167 - Eine binäre ODER-Verknüpfung

Nach dem Start über

G4000

schaut es wie folgt aus.

```
.G4000
  PC   SR AC XR YR SP   NV-BDIZC
;4005 B0 F5 00 00 F6   10110000
.
```



Dazu muss man sich das Ganze am besten wieder auf Bitebene ansehen.

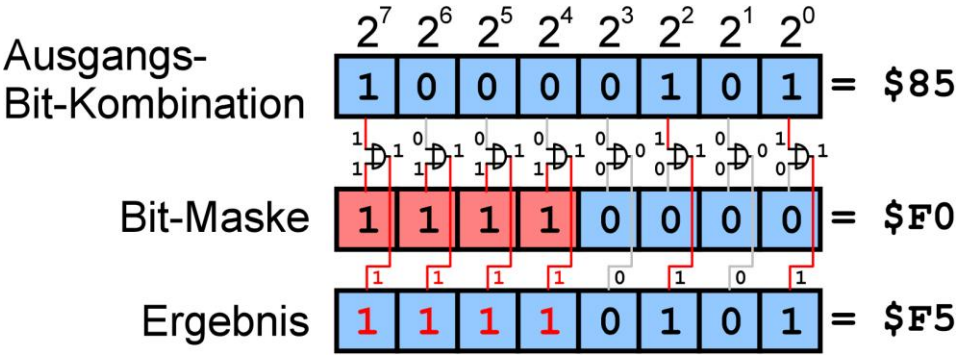


Abbildung 168 - ODER-Verknüpfung von $85 und $F0

Ich denke, dass das jetzt besser verständlich ist, wenn man sich die *ODER*-Gatterverknüpfung der einzelnen Bits anschaut. Überall dort, wo sich in der Bitmaske eine *1* befindet, wird auf jeden Fall im Ergebnis eine *1* zu sehen sein. Wo sich eine *0* befindet, wird die Bitkombination des zu manipulierenden Wertes übernommen. Eine *0* bleibt eine *0* und *1* eine *1* bleibt eine *1*.

Bits löschen

Von der eben gezeigten Bit-Kombination sollen die folgenden drei gelöscht werden.

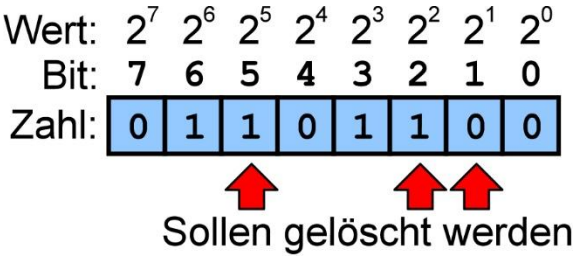


Abbildung 169 - Ausgangs-Bit-Kombination für das Löschen von Bits

Zwei von den Bits sind gesetzt (Bit *2* und *5*) und ein Bit (Bit *1*) ist schon gelöscht. Um ein Bit an einer bestimmten Position zu löschen, kann nicht mit einer *ODER*-Verknüpfung erreicht werden, denn wenn eine *1* vorhanden ist, kann sie nicht mit einer zweiten *0* auf *0* gesetzt werden. Dazu muss die *UND*-Verknüpfung (*AND*-Verknüpfung) zur Anwendung kommen. Nachfolgend ist noch einmal die Wahrheitstabelle einer UND-Verknüpfung zu sehen.

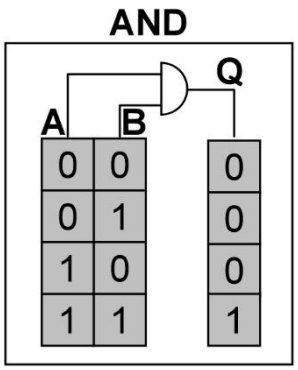


Abbildung 170 - Eine UND-Verknüpfung

Es ist zu sehen, dass beide Eingänge eine *1* vorweisen müssen, damit der Ausgang auf *1* geht. Eine Bit-Maske, die das Löschen von Bits bewirkt, schaut wie folgt aus. Alle Bits, die gelöscht werden sollen, müssen dabei in der Bit-Maske mit einer *0* versehen sein, die unverändert bleiben sollen, müssen in der Bit-Maske mit einer *1* gekennzeichnet sein. Also genau umgekehrt zum Setzen von Bits.

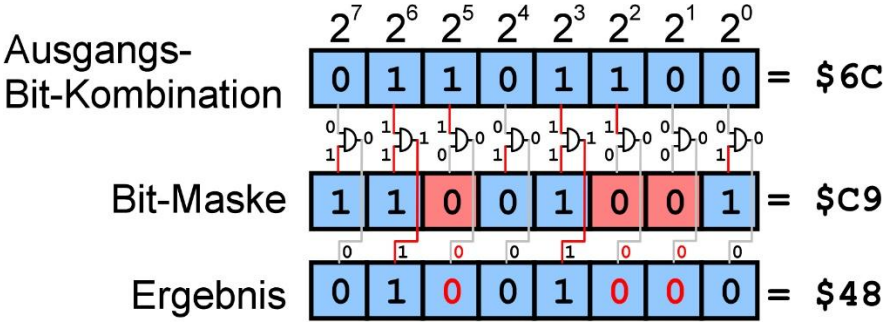


Abbildung 171 - Einzelne Bits werden gelöscht

Machen wir doch ein kleines Experiment für das Löschen der oberen *4* Bits eines Bytes, so dass nur die unteren *4* Bits als Ergebnis übrigbleiben. Ausgangswert ist hierbei *$37* und die Maske *$F0*.

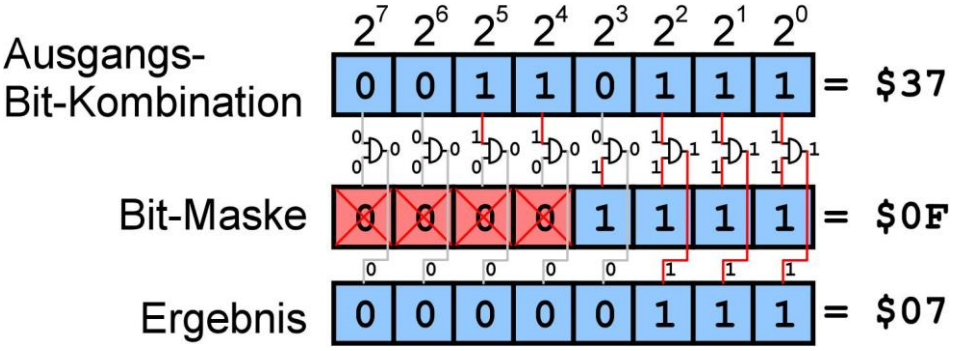


Abbildung 172 - Die obersten 4 Bits werden gelöscht

Das Assembler-Programm dazu schaut wie folgt aus. Genutzt wird der *ORA*-Befehl.

Nachfolgend die Matrix für den *AND*-Befehl.

OP-Code: 29

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,X)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,X)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2

Abbildung 173 - Die OP-Code-Matrixtabelle für den Op-Code AND

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 26 - Die Flag-Beeinflussung des AND-Befehls

Operation: A ∧ M → A (Der AND-Befehl überträgt den Speicher/Wert und den Akkumulator an den Addierer, der eine binäre UND-Verknüpfung durchführt und das Ergebnis im Akkumulator speichert.)

Hier nun das Programm.

```
.D4000
,4000   A9 37       LDA #37
,4002   29 0F       AND #0F
,4004   00          BRK
--------------------------
.
```

Abbildung 174 - Eine binäre UND-Verknüpfung

Nach dem Start über

G4000

schaut es wie folgt aus.

```
.G4000
  PC  SR AC XR YR SP  NV-BDIZC
;4005 30 07 00 00 F6  00110000
.
```

Was mir überhaupt nicht klar ist, warum man den Operanden Bit-Maske nennt. Was hat das Ganze mit einer *Maske* zu tun?

Das ist eine berechtigte Frage und ich zeige dir das am besten anhand einer Grafik. Dort ist der Operand des *AND*-Befehls mit einer Art Lochmaske zu sehen. Überall dort, wo sich an einer Bitposition eine *1* befindet, ist ein Loch in einer - sagen wir mal - Blende, die eben als Maske arbeitet und nur an diesen Stellen etwas durchlässt.

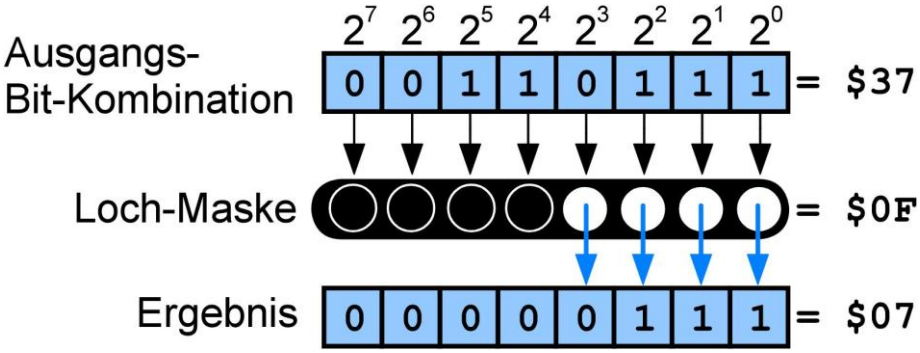


Abbildung 175 - Die Arbeitsweise einer Bit-Maske

Bits toggeln - invertieren

Der Ausdruck *toggeln* kann mit *kippen* übersetzt werden, was bedeutet, dass ein vorheriger logischer Zustand in seinen anderen kippt, also wechselt. Aus *0* wird *1* und aus *1* wird *0*. Natürlich kommt hier wieder eine schöne logische Verknüpfung zum Einsatz, die sich *Exklusiv-ODER* (XOR-Verknüpfung) nennt. Die entsprechende Wahrheitstabelle schaut wie folgt aus.

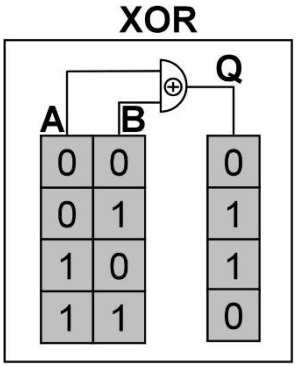


Abbildung 176 - Eine XOR-Verknüpfung

Der Ausgang ist immer dann *1*, wenn beide Eingänge unterschiedliche Pegel vorweisen. Wie aber sollte man eine derartige Verknüpfung mit welchen Werten herstellen? Nun, sehen wir uns dazu die Wahrheitstabelle genauer an und zwar genau an den Stellen, an denen der zweite Eingang *B* gleich *1* ist. Ist der erste Eingang *A* gleich *0* ist das Ergebnis *1* und ist er *1* lautet das Ergebnis *0*. Es erfolgt also eine Umkehrung des logischen Pegels. Um nun ein bestimmtes Bit umzukehren, ist einfach eine *XOR*-Verknüpfung mit *1* erforderlich. Sehen wir uns das am folgenden Beispiel genauer an, wobei wiederum die zuvor genutzte Bit-Kombination zur Anwendung kommt.

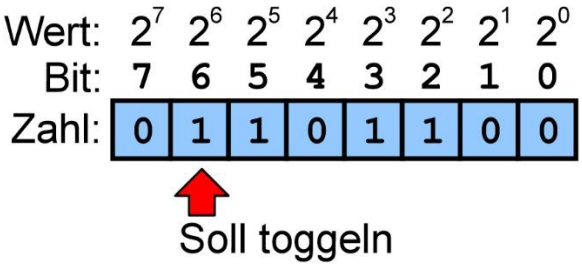


Abbildung 177 - Ausgangs-Bit-Kombination für das Toggeln eines Bits

Also werde ich - für das bessere Verständnis - die *XOR*-Verknüpfung von Bit *6* zweimal hintereinander mit dem Wert *1* durchführen. Nachfolgend mit *1.Ergebnis* und *2.Ergebnis* markiert.

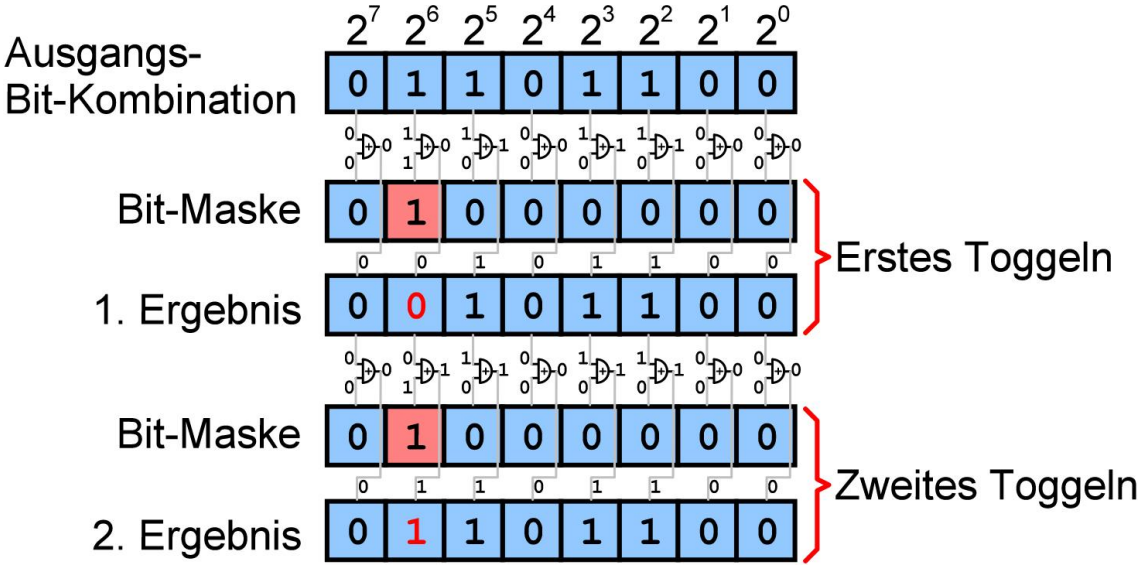


Abbildung 178 - Zweimaliges Toggeln von Bit 6

Es ist zu sehen, dass sich Bit *6* beim ersten Mal von *1* auf *0* ändert und beim *2*. Mal von *0* auf *1* zurück. Wenn man alle in der LED-Anzeige aufleuchtenden Bits umkehren möchte, ist eine *XOR*-Verknüpfung mit dem Wert *$FF* erforderlich.

Genutzt wird der *ORA*-Befehl. Nachfolgend die Matrix für den *EOR*-Befehl.

OP-Code: 49

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4

Abbildung 179 - Die OP-Code-Matrixtabelle für den Op-Code EOR

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

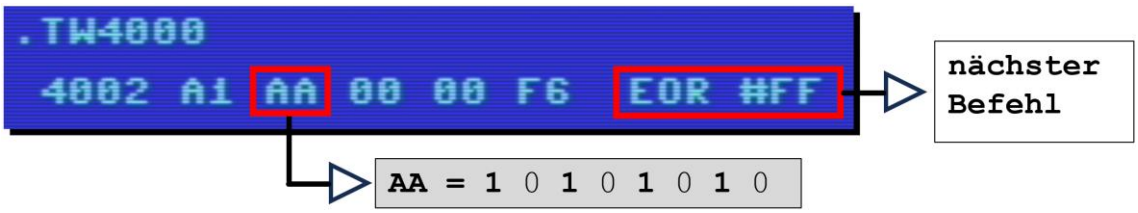
Tabelle 27 - Die Flag-Beeinflussung des EOR-Befehls

Operation: A ⊻ M → A (Der AND-Befehl überträgt den Speicher/Wert und den Akkumulator an den Addierer, der eine binäre EXKLUSIV-ODER-Verknüpfung durchführt und das Ergebnis im Akkumulator speichert.)

Das folgende Programm soll den Inhalt des Akkus, der binär *10101010* (*$AA*) lautet über den *EOR*-Befehl schrittweise toggeln. Hier nun das Programm, das wir dann im *Trace-Walk*-Modus mit der Eingabe von

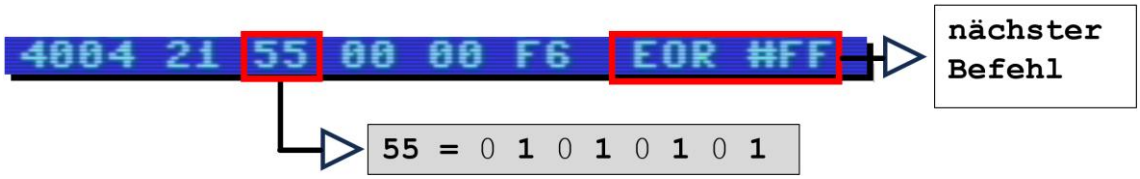
TW4000

starten werden. Es ist zu sehen, dass der Akku mit dem Wert *$AA* geladen wurde, was der schon genannten Bitkombination *10101010* entspricht.



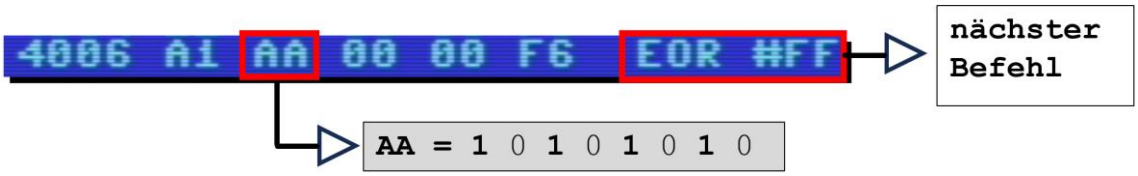
Drücken wir jetzt 1x die Leertaste.

Nach dem erstmaligen Ausführen des *EOR*-Befehls mit dem Operanden *$FF* wurden alle Bits invertiert, so dass jetzt der Wert *$55* im Akku zu sehen ist, was der Bitkombination *01010101* entspricht.



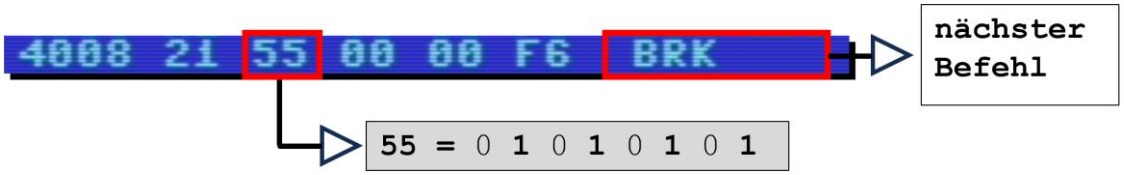
Drücken wir jetzt 1x die Leertaste.

Nach dem erneuten Ausführen des *EOR*-Befehls mit dem Operanden *$FF* wurden wieder alle Bits invertiert, so dass jetzt der ursprüngliche Wert *$AA* im Akku zu sehen ist, was natürlich der Bitkombination *10101010* entspricht.



Drücken wir jetzt 1x die Leertaste.

Nach einem weiteren Ausführen des *EOR*-Befehls mit dem Operanden *$FF* wurden - wie sollte es anders sein - wieder alle Bits invertiert, so dass jetzt der Wert *$55* im Akku zu sehen ist, was der Bitkombination *01010101* entspricht.



Der nächste auszuführende *BRK*-Befehl beendet den Programmablauf.

Bits schieben

Ja, nicht nur in höheren Posten gibt es Schiebereien, sondern auch in der CPU. Es gibt bei der ganzen Schieberei einiges zu beachten und auch ein Rotieren von Bits ist möglich. Ich fange ganz einfach mit dem *nach-rechts-schieben* von Bits an.

Nach rechts schieben

Der Befehl für das nach-rechts-schieben lautet *LSR*, was die Abkürzung für (***L**ogical-**S**hift-**R**ight*) ist. Sehen wir uns den entsprechenden OP-Code zum *LSR*-Befehl an.

OP-Code: 4A

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4

Abbildung 180 - Die OP-Code-Matrixtabelle für den Op-Code LSR

Das ***A*** unterhalb des Befehls deutet daraufhin, dass der Akku bei dieser Operation betroffen ist. Es gibt aber auch einen OP-Code, der sich auf eine bestimmte Speicherstelle auswirkt. Dieser Befehl verschiebt den Inhalt des Akkumulators um 1 Bit nach rechts, wobei das höhere Bit des Ergebnisses immer auf *0* gesetzt wird (das Ergebnis ist demnach immer positiv zu interpretieren) und das niedrige Bit, das quasi rechts herausfliegt, im Carry-Flag landet.

N	V	-	B	D	I	Z	C
0	-	-	-	-	-	✓	✓

Tabelle 28 - Die Flag-Beeinflussung des LSR-Befehls

Operation: 0 → /M_7...M_0/ → C (Der Akku-Inhalt wird nach rechts verschoben)

Auf der folgenden Abbildung ist das allgemeine Verhalten zu sehen.

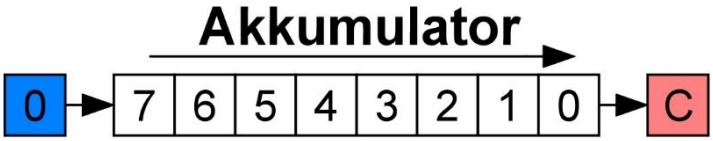


Abbildung 181 - Der LSR-Befehl (Auswirkung auf den Akkumulator)

Sehen wir uns das konkret an einem Beispiel mit dem Wert *$B5* (*181*) an.

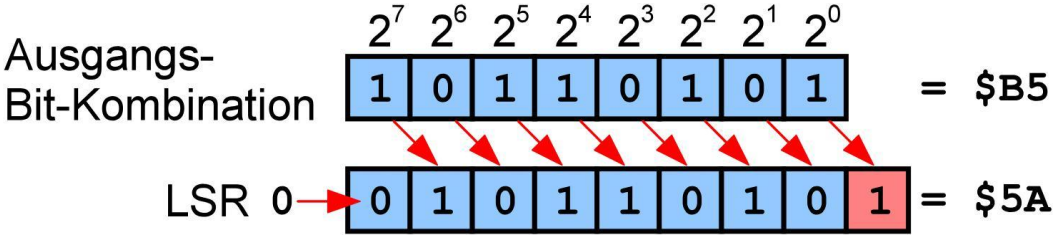


Abbildung 182 - Einmaliges Schieben der Bits nach rechts

Das Programm schaut wie folgt aus.

```
  PC   SR AC XR YR SP   NV-BDIZC
;4102  32 0D 18 00 DE   00110010
.D4000
,4000   18          CLC
,4001   A9 B5       LDA #B5
,4003   4A          LSR
,4004   00          BRK
--------------------------------
.
```

Abbildung 183 - Der LSR-Befehl

Nach dem Start über

G4000

ist im Akku der Wert *$5A* zu sehen. Das niederwertigste Bit 0 ist hier in das Carry-Flag gewandert und hat es auf *1* gesetzt.

```
.G4000
  PC   SR AC XR YR SP   NV-BDIZC
;4005  31 5A 18 00 DE   00110001
.
```

Abbildung 184 - Der LSR-Befehl wurde auf den Akku-Inhalt angewendet

Wir können ja mal ein Schiebeprogramm schreiben, das mehrere *LSR*-Schiebebefehle auf den Wert *#$37* hintereinander ausführt.

```
,4000  18        CLC
,4001  A9 37     LDA #37
,4003  4A        LSR
,4004  4A        LSR
,4005  4A        LSR
,4006  4A        LSR
,4007  00        BRK
--------------------------
.
```

Abbildung 185 - Ein vierfaches LSR

Nach dem Start über

G4000

wurde der Wert im Akku angepasst.

```
.G4000
  PC  SR AC XR YR SP  NV-BDIZC
;4008 30 03 00 00 F6  00110000
.
```

Abbildung 186 - Aus dem Wert $37 wurde $03

Stimmt das auch? Sehen wir nach, wie der Wert *$03* entstanden ist.

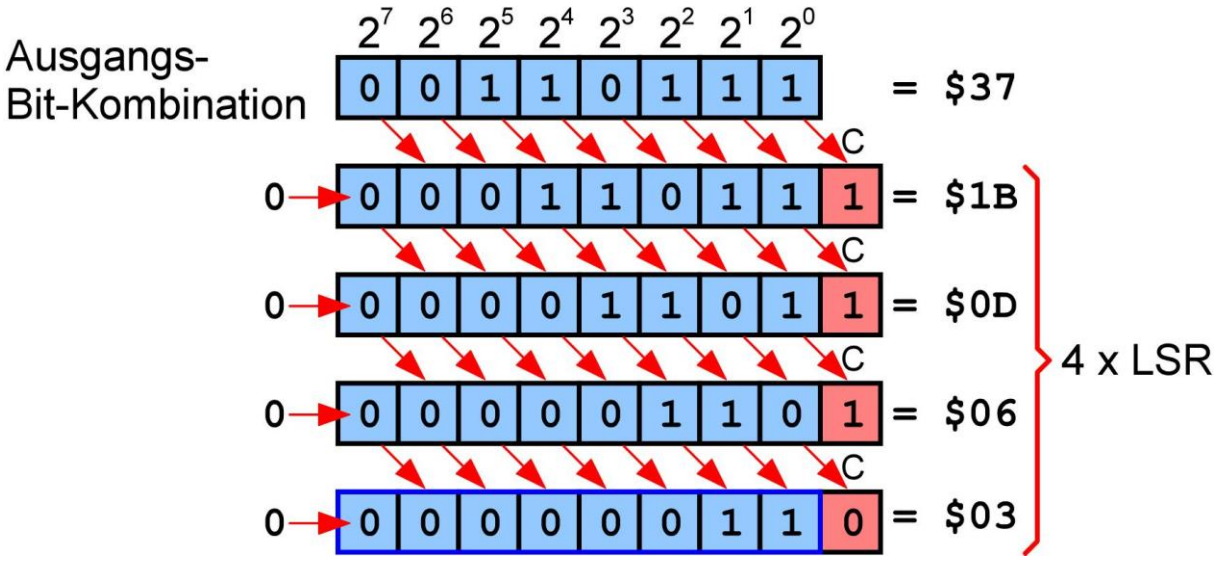


Abbildung 187 - Der Weg wie aus $37 der Wert $03 wurde

Es ist auch zu sehen, dass bei der letzten Schiebeaktion das Carry-Flag nicht gesetzt wurde, was bei den vorherigen dreien aber der Fall war. Ich würde vorschlagen, die einzelnen Schritte mit einem *Trace-Walk* abzuarbeiten.

Nach links schieben

Der Befehl für das nach-links-schieben lautet *ASL*, was die Abkürzung für (***A**rithmetic-**S**hift-**L**eft*) ist. Sehen wir uns den entsprechenden OP-Code zum *ASL*-Befehl an. Nachfolgend die Matrix für den *ASL*-Befehl.

Sehen wir uns den entsprechenden OP-Code zum *ASL*-Befehl an.

OP-Code: 0A

MSD	W65C02S OpCode Matrix															MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,X)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0

Abbildung 188 - Die OP-Code-Matrixtabelle für den Op-Code ASL

Das ***A*** unterhalb des Befehls deutet wiederum daraufhin, dass der Akku bei dieser Operation betroffen ist. Es gibt aber auch einen OP-Code, der sich auf eine bestimmte Speicherstelle auswirkt. Dieser Befehl verschiebt den Akkumulator um *1* Bit nach links, wobei das Bit *0* immer auf *0* gesetzt und Bit *7* im Carry-Flag landet.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	✓

Tabelle 29 - Die Flag-Beeinflussung des ASL-Befehls

Operation: C ← /M_7...M_0/ ← 0 (Der Akku-Inhalt wird nach links verschoben)

Auf der folgenden Abbildung ist das allgemeine Verhalten zu sehen.

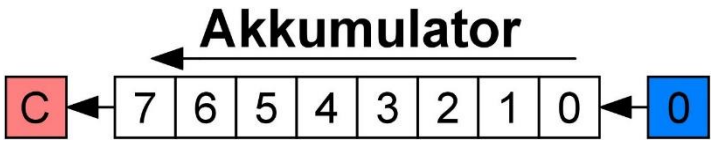


Abbildung 189 - Der ASL-Befehl (Auswirkung auf den Akkumulator)

Sehen wir uns das konkret an einem Beispiel mit dem Wert *$0F* (*15*) an.

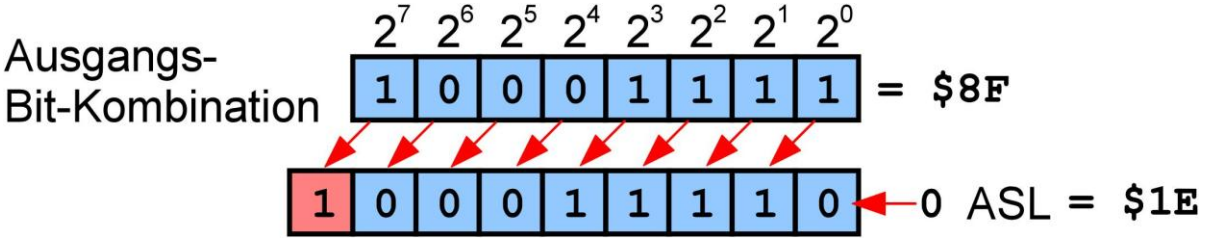


Abbildung 190 - Einmaliges Schieben der Bits nach links

Das Programm schaut wie folgt aus, wobei das Carry-Flag im Moment noch nicht gesetzt ist.

```
  PC   SR AC XR YR SP   NV-BDIZC
;C00B  B0 C2 00 00 F2   10110000
.D4000
,4000   18          CLC
,4001   A9 8F       LDA #8F
,4003   0A          ASL
,4004   00          BRK
--------------------------------
.
```

Abbildung 191 - Der ASL-Befehl

Nach dem Start über

G4000

ist im Akku der Wert *$1E* zu sehen.

```
.G4000
  PC   SR AC XR YR SP   NV-BDIZC
;4005  31 1E 00 00 F2   00110001
.
```

Abbildung 192 - Der ASL-Befehl wurde auf den Akku-Inhalt angewendet

Da Bit 7 - das höchstwertigste Bit - vor dem Schiebevorgang schon gesetzt war, landet es nach dem Linksschieben über *ASL* im Carry-Flag, was jetzt auf *1* gesetzt ist.

Nach rechts rotieren

Wenn etwas rotiert, dann dreht sich etwas um eine Achse. Man kann das im übertragenen Sinn sehen, denn was auf der einen Seite quasi verschwindet, kommt auf der anderen Seite wieder zum Vorschein. Werden Bits beim links oder rechts schieben aus dem Bitmuster herausgeschoben, wandert das äußere Bit zwar in das Carry-Flag und ist noch nicht verloren, doch beim nächsten Mal ist es dann weg. Beim Rotieren nach rechts über den *ROR*-Befehl (***RO****tate* ***R****ight*) wird das herausgeschobene Bit *0* ebenfalls ins Carry-Flag übertragen, wandert beim nächsten Mal jedoch wieder auf der anderen Seite in Bit 7 hinein.

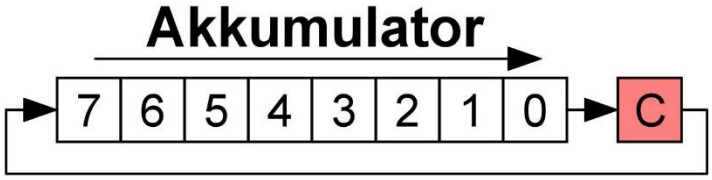


Abbildung 193 - Der ROR-Befehl (Auswirkung auf den Akkumulator)

Sehen wir uns den entsprechenden OP-Code zum *ROR*-Befehl an.

OP-Code: 6A

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r •	6

Abbildung 194 - Die OP-Code-Matrixtabelle für den Op-Code ROR

Das ***A*** unterhalb des Befehls deutet wiederum daraufhin, dass der Akku bei dieser Operation betroffen ist. Es gibt aber auch einen OP-Code, der sich auf eine bestimmte Speicherstelle auswirkt. Dieser Befehl verschiebt entweder den Akkumulator oder den adressierten Speicher um *1* Bit nach rechts, wobei Bit *0* in das Carry-Flag geschoben wird und der dort zuvor vorhandene Wert nach Bit 7 wandert.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	✓

Tabelle 30 - Die Flag-Beeinflussung des ROR-Befehls

Operation: C → /M_7...M_0/ → C (Der Akku-Inhalt wird nach rechts rotiert)

Machen wir doch eine Simulation mit einem Wert, der in vier Schritten nach rechts rotieren soll. Der Ausgangswert lautet hierbei *$85*, was der Bitkombination *10000101* entspricht. Zuerst sehen wir uns die Schritte im Detail an und lassen im Anschluss das Programm im *Trace-Walk*-Modus zur Kontrolle durchlaufen.

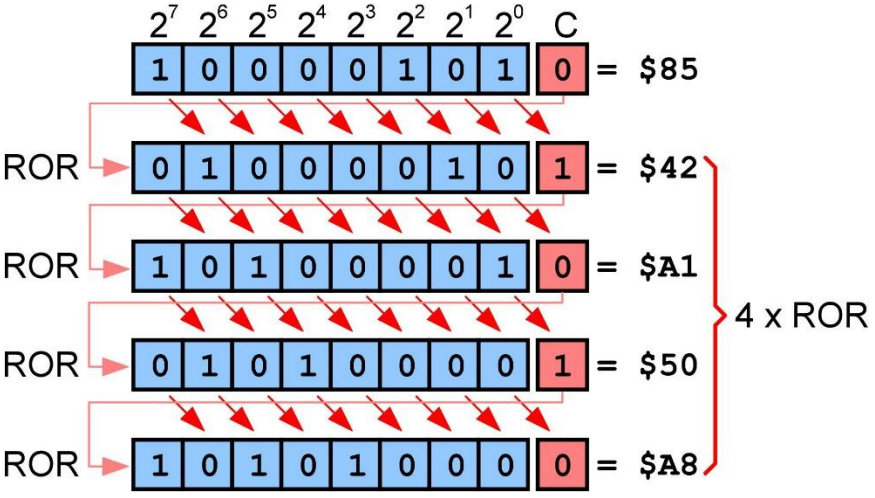


Abbildung 195 - Viermaliges Rotieren nach rechts

Es ist zu sehen, dass der Ausgangswert *$85* schrittweise die Werte *$42*, *$A1*, *$50* und *$A8* durch die Rechtsrotation bildet. Das Programm dazu schaut wie folgt aus.

```
  PC   SR AC XR YR SP   NV-BDIZC
;4009 B0 A8 00 00 F2   10110000
.D4000
,4000   18          CLC
,4001   A9 85       LDA #85
,4003   6A          ROR
,4004   6A          ROR
,4005   6A          ROR
,4006   6A          ROR
,4007   00          BRK
--------------------------------
.
```

Abbildung 196 - Das Programm für das Vierfache Rotieren nach rechts

Starten wir die schrittweise Abarbeitung. Zur besseren Übersicht habe ich rechts neben dem Programm noch das Status-Register *SR* mit seinen Werten gezeigt, so dass der Inhalt des Carry-Flags besser zu erkennen ist.

```
.TW4000

 4001 A0 A8 00 00 F2   LDA #85
 4003 A0 85 00 00 F2   ROR
 4004 21 42 00 00 F2   ROR
 4005 A0 A1 00 00 F2   ROR
 4006 21 50 00 00 F2   ROR
 4007 A0 A8 00 00 F2   BRK

  PC  SR AC XR YR SP   NV-BDIZC
;4009 B0 A8 00 00 F2   10110000
.
```

```
SR                C
A0 = 10100000
21 = 00100001
A0 = 10100000
21 = 00100001
A0 = 10100000
```

Abbildung 197 - Die schrittweise Abarbeitung des ROR-Befehls

Nach links rotieren

Bei der Rotation nach links über den *ROL*-Befehl (***RO**tate **L**eft*) verhält es sich genau umgekehrt wie zuvor beim *ROR*-Befehl. Beim Rotieren wird das herausgeschobene Bit *7* ebenfalls ins Carry-Flag übertragen, wandert beim nächsten Mal jedoch wieder auf der anderen Seite in Bit *0* hinein.

Abbildung 198 - Der ROL-Befehl (Auswirkung auf den Akkumulator)

Sehen wir uns den entsprechenden OP-Code zum *ROR*-Befehl an.

OP-Code: 2A

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2

Abbildung 199 - Die OP-Code-Matrixtabelle für den Op-Code ROL

Das ***A*** unterhalb des Befehls deutet wiederum daraufhin, dass der Akku bei dieser Operation betroffen ist. Es gibt aber auch einen OP-Code, der sich auf eine bestimmte Speicherstelle auswirkt. Dieser Befehl verschiebt entweder den Akkumulator oder den adressierten Speicher um *1* Bit nach links, wobei Bit *7* in das Carry-Flag geschoben wird und der dort zuvor vorhandene Wert nach Bit 0 wandert.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	✓

Tabelle 31 - Die Flag-Beeinflussung des ROL-Befehls

Operation: C ← /M_7...M_0/ ← C (Der Akku-Inhalt wird nach links rotiert)

Eine Zusammenfassung - LSR, ASL, ROR, ROL

Um einen Überblick aller vier genannten Bit-Manipulationen zu bekommen, habe ich diese in der folgenden Grafik zusammengefasst.

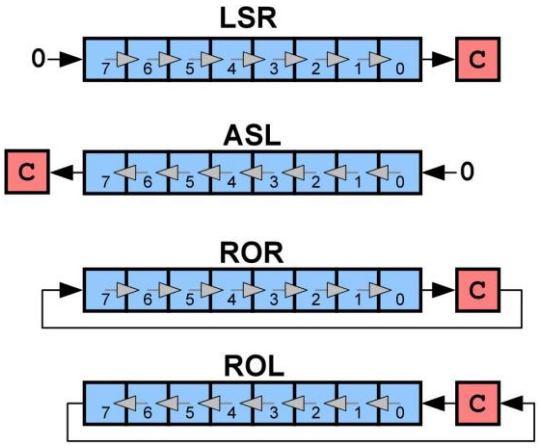


Abbildung 200 - Alle vier Bit-Manipulationen auf einen Blick

Ein weiteres Zahlensystem

- Der BCD-Code

Ein weiteres Zahlensystem

Da wir es mit einem Computer zu tun haben, der auf unterster Ebene nur Informationen versteht, die aus Nullen und Einsen zusammengesetzt sind, wurde das Binärsystem entwickelt. Das haben wir jetzt schon kennengelernt. Ich möchte nun auf ein weiteres Zahlensystem eingehen, das sich *BCD* nennt.

Der BCD-Code

Der sogenannte *BCD*-Code bedeutet übersetzt *Binary Coded Decimal*, also dualkodierte Dezimalziffer. Dieser Kode ist mit dem binären Zahlensystem eng verwandt. Im BCD-Code wird jede Dezimalziffer durch vier binäre Stellen - also *4*-Bit - dargestellt. Eine Einheit dieser *4*-Bits wird entweder *Nibble* oder *Tetrade* (griechisch: Vierergruppe) genannt. Das folgende Beispiel zeigt eine *3*-Stellige Dezimalzahl und die entsprechende BCD-Kodierung.

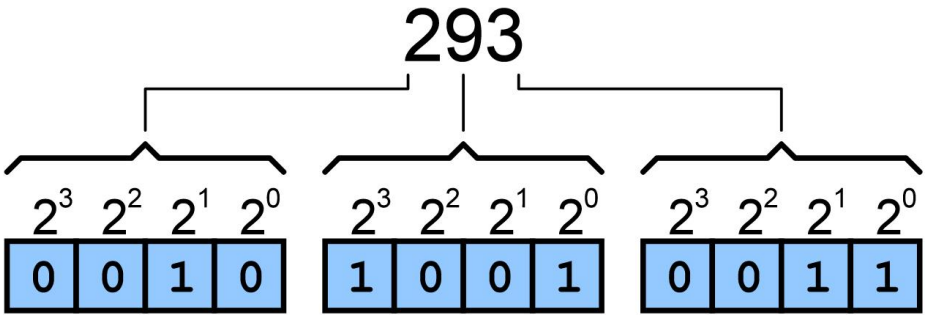


Abbildung 201 - Eine Dezimalzahl im BCD-Code

Es ist zu sehen, dass jede einzelne Ziffer der Dezimalzahl *293* als eine Binärzahl von *4*-Bits ausgedrückt wird. Nun gibt es im Dezimalsystem bekanntlich *10* unterschiedliche Ziffern (0 bis 9). Im Gegensatz dazu lassen sich mit *4*-Bits 2^4, also *16* Werte darstellen. Von den möglichen *16* zur Verfügung stehenden Tetraden werden nur *10* genutzt. Die restlichen sechs dürfen im BCD-Code nicht verwendet werden und erhalten die Bezeichnung *Pseudo-Tetraden*.

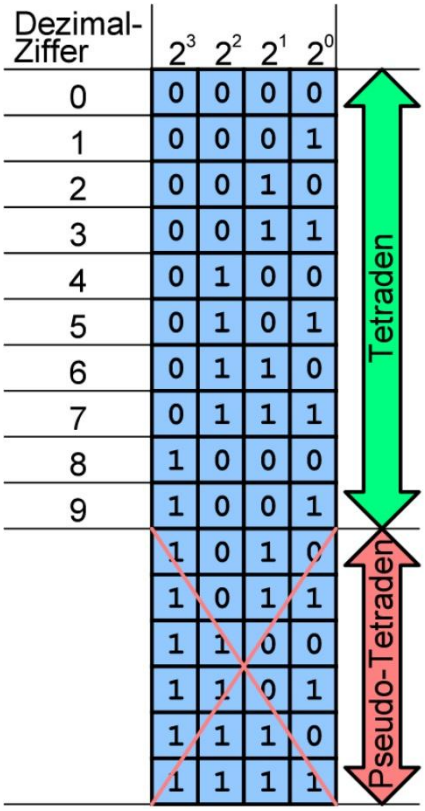


Abbildung 202 - Der BCD-Code

Hierzu ein zwei Beispiele mit Additionen, die unproblematisch sind, wenn das Ergebnis nicht in den unteren Bereich (Ergebnis > *9*) der Pseudo-Tetraden fällt.

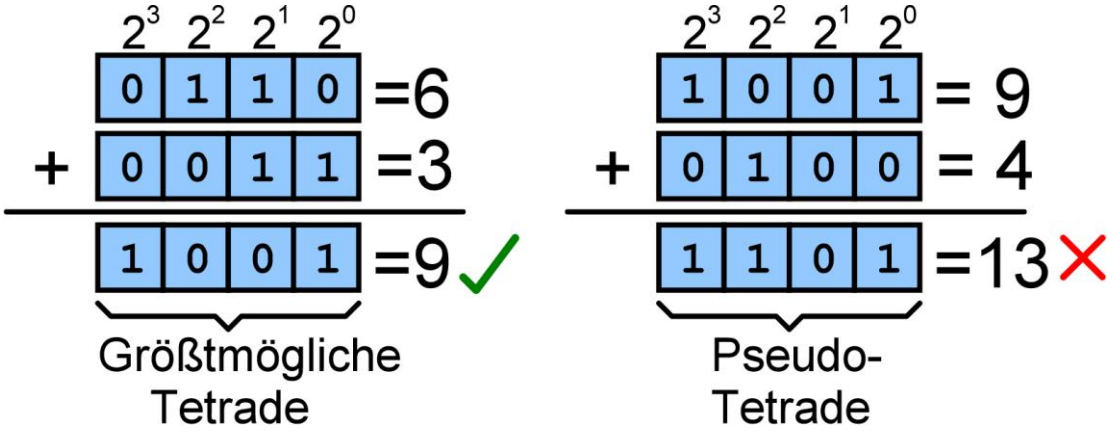


Abbildung 203 - Korrekter und unerlaubter BCD-Code

Auf der linken Seite besteht die Addition aus den Werten *6* und *3*, was im Ergebnis noch eine erlaubte Tetrade *9* ergibt. Dagegen wird auf der rechten Seite das Ergebnis der Addition *9* und *4* zu einer Pseudo-Tetrade führen, da das Ergebnis *13* zweistellig ist und deswegen einer Korrektur über die Addition mit dem Wert *+6* bedarf, wie das auf der folgenden Abbildung zu erkennen ist.

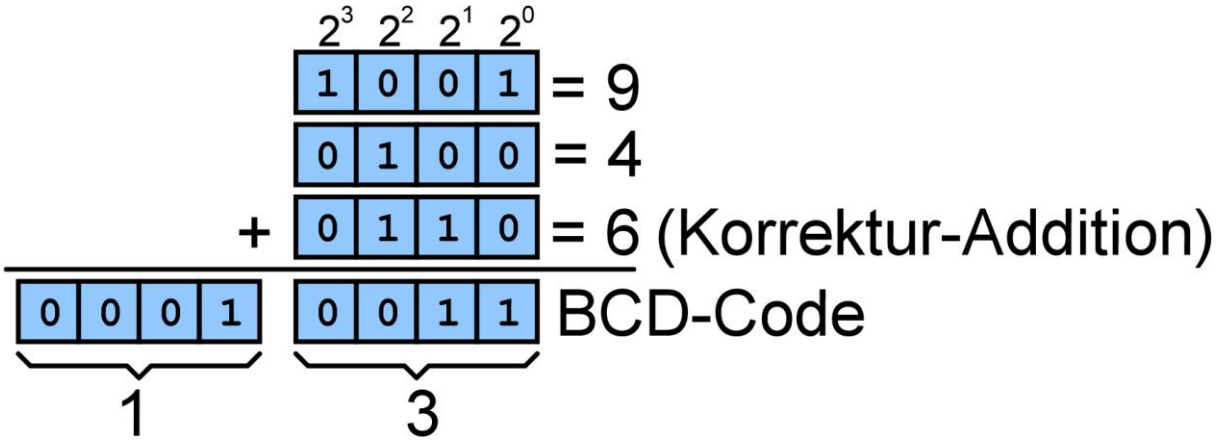


Abbildung 204 - Die Korrektur der Pseudo-Tetrade über die Addition mit dem Wert +6

An dem Ergebnis ist zu erkennen, dass die Zehner-Stelle in eine neue Tetrade gewandert ist. Wenn die Addition von BCD-Zahlen aus mehreren Tetraden vorliegt, dann ist die Addition Tetraden Weise von rechts nach links durchzuführen, wie das auf der folgenden Abbildung zu sehen ist. Dabei ist die Korrektur durch die Addition von *+6* immer dann vorzunehmen, wenn das Ergebnis der Addition zweiter Tetraden gleich oder größer *10* ist.

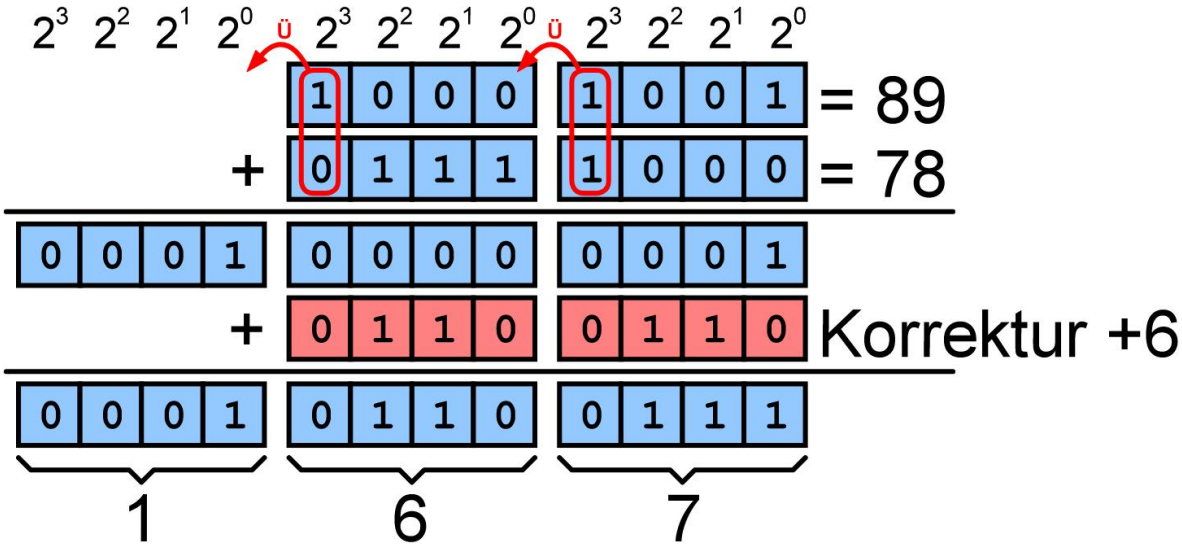


Abbildung 205 - BCD-Addition mit Korrekturen

Der in der Abbildung gezeigte Übertrag *Ü* bezieht sich nur auf den Tetraden-Übertrag, wobei natürlich pro Tetrade ebenfalls Überträge an unterschiedlichen Stellen vorkommt.

Sehen wir uns das Verhalten einmal in einem Assembler-Programm an, denn die *6502-/6510*-CPU besitzt einen speziellen *BCD*-Modus, der über OP-Codes aktiviert *SED* (***SE****t* ***D****ecimal*) und deaktiviert *CLD* (***CL****ear* ***D****ecimal*) werden kann.

D	BNE r	CMP (zp),y	CMP (zp) *			CMP zp,x	DEC zp,x	SMB5 zp •	CLD i	CMP a,y	PHX s •	STP I •		CMP a,x	DEC a,x	BBS5 r •	D
E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX	SBC #	NOP i		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ r	SBC (zp),y	SBC (zp) *			SBC zp,x	INC zp,x	SMB7 zp •	SED i	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 206 - Die OP-Code-Matrixtabelle für die Op-Codes SED und CLD

Auf der nachfolgenden Abbildung ist das D-Flag hervorgehoben.

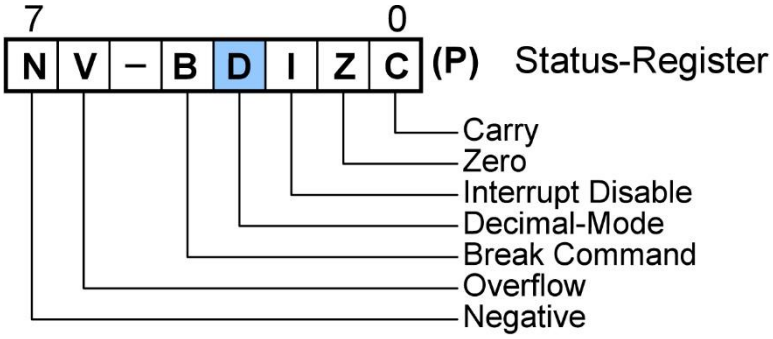


Abbildung 207 - Das Status-Register – Das Dezimal-Flag

Der *SED*-Befehl wirkt sich direkt auf das D-Flag (Dezimal) aus.

N	V	–	B	D	I	Z	C
–	–	–	–	1	–	–	–

Tabelle 32 - Die Flag-Beeinflussung des SED-Befehls

Operation: 1 → D (Wert *1* wird ins D-Flag übertragen)

Durch diesen Befehl werden alle nachfolgenden *ADC*- und *SBC*-Befehle als dezimale arithmetische Operation betrachtet.

Der *CLD*-Befehl wirkt sich ebenfalls direkt auf das D-Flag aus.

N	V	–	B	D	I	Z	C
–	–	–	–	0	–	–	–

Tabelle 33 Die Flag-Beeinflussung des CLD-Befehls

Operation: 0 → D (Wert *0* wird ins D-Flag übertragen)

Durch diesen Befehl werden alle nachfolgenden *ADC*- und *SBC*-Anweisungen als einfache Operationen durchgeführt und wie bisher behandelt.

Zuerst fange ich mit einem Assembler-Programm an, das so arbeitet, wie wir das bisher gewohnt waren, wobei ich eine einfache *5 + 5* Addition durchführe.

Das Assembler-Programm schaut wie folgt aus.

```
,4000  A9 05     LDA #05
,4002  18        CLC
,4003  69 05     ADC #05
,4005  00        BRK
--------------------------
.G4000

  PC  SR AC XR YR SP  NV-BDIZC
;4006 30 0A 00 00 F6  00110000
.
```

Abbildung 208 - Eine normale Addition

Das Ergebnis der Addition der beiden Werte ergibt im Akku *$A0*, was für den dezimalen Wert *10* steht. Also wirklich sehr einfach.

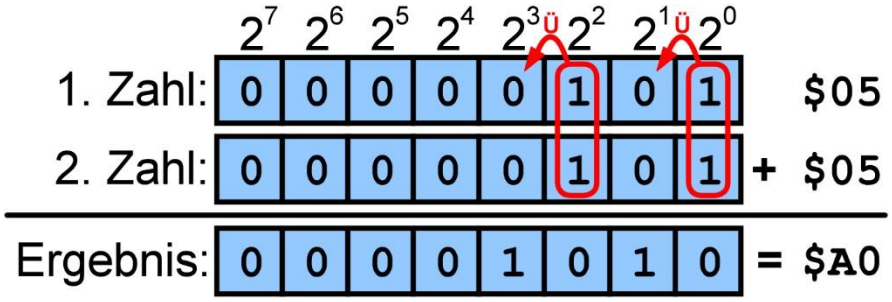


Abbildung 209 - Die Addition $05 + $05

Sehen wir uns dazu das folgende Programm an, bei dem über den *SED*-Befehl der BCD-Mode aktiviert wurde.

```
,4000  F8        SED
,4001  A9 05     LDA #05
,4003  18        CLC
,4004  69 05     ADC #05
,4006  00        BRK
--------------------------
.G4000

  PC  SR AC XR YR SP  NV-BDIZC
;4007 38 10 00 00 F6  00111000
.
```

Abbildung 210 - Eine BCD-Addition

Das Ergebnis der Addition der beiden Werte im BCD-Mode ergibt im Akku jetzt aber *$10*.

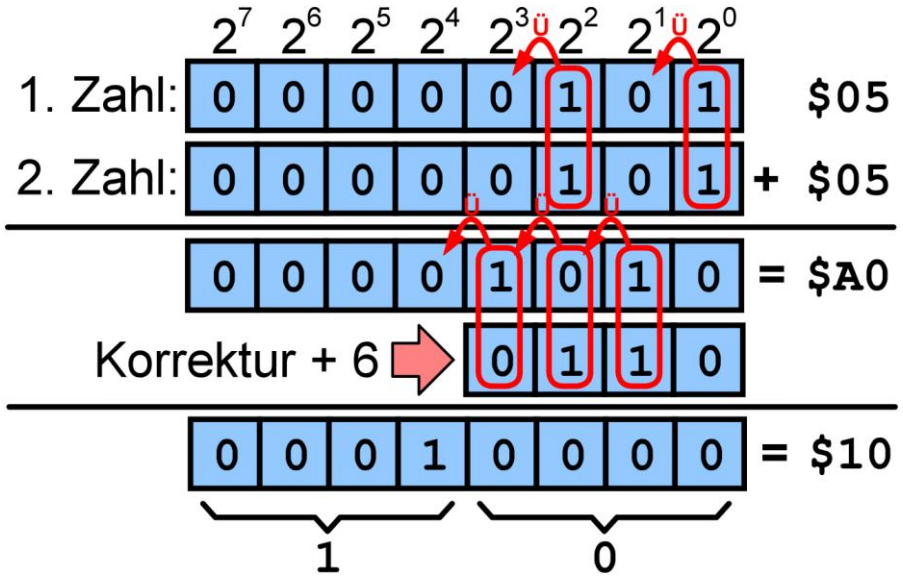


Abbildung 211 - Die Addition $05 + $05 im BCD-Mode

Das Ergebnis *$10* muss nun nicht als ganzer Wert betrachtet werden, sondern stellenweise *1* und *0*.

Fertige Routinen

- Die CHROUT-Routine - Teil 1
- Die CHROUT-Routine - Teil 2
- Die GETIN-Routine
- Die Sache mit dem Stapel - Teil 1
- Die Sache mit dem Stapel - Teil 2

Fertige Routinen

Der C64 bietet in seinem ROM fertige Routinen - in Form von Unterprogrammen - an, die man nutzen kann. Angenommen, man möchte auf das Diskettenlaufwerk zugreifen oder etwas auf dem Bildschirm ausgeben, so gibt es dafür fertige Maschinenprogramme, die an bestimmten Adressen im Speicher liegen. Natürlich ist es dann gut zu wissen, wie die Einsprungadressen lauten und welche Parameter vorher zu setzen sind. Bleiben wir doch einfach einmal bei der Bildschirmausgabe. Derartige Routinen sind in der Regel mit entsprechenden Kürzeln versehen und so auch in diesem Fall. Die drei nachfolgenden Routinen werden immer wieder benötigt und aus diesem Grund werde ich sie vorstellen. Es gibt natürlich sehr viel mehr, doch das würde dieses kleine Büchlein etwas überfordern.

Name	Einsprungadresse	Funktion
CHROUT	$FFD2 (65490)	Gibt ein Zeichen an der aktuellen Cursorposition aus
GETIN	$FFE4 (65508)	Liest ein Zeichen
STOP	$FFE1 (65505)	Überprüft die *RUN/STOP*-Taste (in VICE ist es die *ESC*-Taste)

Tabelle 34 - Drei wichtige System-Routinen

Über den nachfolgenden Link kann man sich über ein paar System-Routinen, die auch *Kernal*-Routinen genannt werden, informieren.

Kernal-Routinen

- https://www.c64-wiki.de/wiki/KERNAL
- https://skoolkid.github.io/sk6502/c64rom/maps/routines.html

Die CHROUT-Routine - Teil 1

Die sogenannte *Output Subroutine* besitzt den Namen *CHROUT*, was für *Character Output* steht und ein Zeichen an der aktuellen Cursorposition ausgibt. Die Funktionalität ist vergleichbar mit einem *PRINT*-Befehl. Welches Zeichen das ist, hängt von Inhalt des Akkus ab, der dann vor dem Aufruf initialisiert werden muss. Diese Adresse lautet *$FFD2* (*65490*). Dann wollen wir doch einmal ein kleines Assemblerprogramm schreiben, das dieses Unterprogramm nutzt und ein Zeichen ausgibt.

```
.D4000
,4000  A9 41     LDA #41
,4002  20 D2 FF  JSR FFD2
,4005  00        BRK
--------------------------
.
```

Abbildung 212 - Der Aufruf der CHROUT-Routine

Der Akku wird mit dem Wert *$41* geladen was dezimal *65* entspricht und das sollte der Wert für den Buchstaben ***A*** sein. Starten wir das Programm über

G4000

und sehen, was passiert.

```
.G4000A
  PC  SR AC XR YR SP  NV-BDIZC
;4006 30 41 00 00 F6  00110000
.
```

Abbildung 213 - Ein A erscheint an der vorherigen Cursor-Position

Es ist zu sehen, dass sich hinter *G000* jetzt ein ***A*** befindet. Dort befand sich der blinkende Cursor vor der Bestätigung über die *RETURN*-Taste.

```
.G4000
```

Das Betriebssystem kennt natürlich die aktuelle Cursor-Position und nutzt diese dann für weitere Verarbeitungen wie in diesem Fall.

Machen wir doch einen weiteren Test und modifizieren das kleine Programm, in dem wir den *BRK*-Befehl durch einen *RTS*-Befehl ersetzen.

Was sollte das denn für einen Sinn ergeben? Das Assemblerprogramm wird dadurch beendet, wie das auch beim *BRK*-Befehl der Fall war!?

Nun, das stimmt nicht ganz. Ich ersetze *BRK* durch *RTS*, um das kleine Maschinenprogramm aus Basic heraus als Unterprogramm aufzurufen. Nach der Abarbeitung erfolgt dann der Rücksprung über *RTS* an die aufrufende Instanz und das ist Basic. Aber sehen wir uns das genauer an, denn hier arbeiten mehrere Unterprogramme quasi Hand in Hand. Das folgende Bild soll die unterschiedlichen Aufrufe für das geplante BASIC-Programm zeigen.

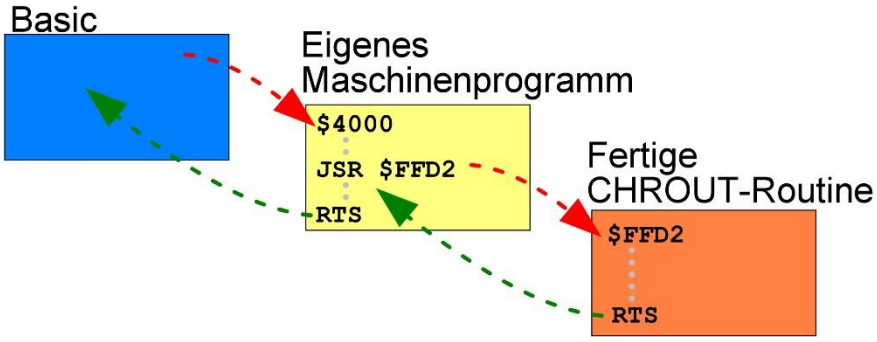


Abbildung 214 - Die Aufrufe der Unterprogramme

Wenn ein Unterprogramm über den *JSR*-Befehl (***J**ump to **S**ub**R**outine*) aufgerufen wird, dann soll dieses nach der Abarbeitung über den *RTS*-Befehl verlassen werden, um somit die Kontrolle an den Aufrufer zurückzugeben. Die Programmausführung fährt dann mit dem Befehl fort, der unmittelbar hinter dem *JSR*-Befehl zu finden ist.

Nachfolgend die Matrix für den *JSR*-Befehl.

OP-Code: 20

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2

Abbildung 215 - Die OP-Code-Matrixtabelle für den Op-Code JSR

Das ***a*** unterhalb des *JSR*-Befehls bedeutet, dass es sich um eine absolute Adressierung handelt. Es werden durch diesen Befehl keine Flags beeinflusst.

N	V	-	B	D	I	Z	C
-	-	-	-	-	-	-	-

Tabelle 35 - Die Flag-Beeinflussung des JSR-Befehls

Operation: PC + 2↓, [PC + 1] → PCL, [PC + 2] → PCH

Dieser Befehl überträgt die Kontrolle über den Programmzähler an eine Unterprogrammstelle und hinterlässt einen Rücksprungzeiger auf dem Stack, damit die nächste Anweisung im Hauptprogramm ausgeführt werden kann, nachdem das Unterprogramm abgeschlossen wurde. Dazu speichert der *JSR*-Befehl die Adresse des Programmzählers, die auf das letzte Byte der Sprunganweisung zeigt, mit Hilfe des Stack-Pointers auf dem Stack. Das Stack-Byte enthält zuerst das High-Byte des Programmzählers, gefolgt vom Low-Byte des Programmzähler. Dazu später mehr. Kommen wir jetzt zu unsrem Vorhaben und sehen uns noch einmal das zuvor

geschriebene Assemblerprogramm an, was - wie schon erwähnt - leicht modifiziert wurde, um einen Rücksprung zu gewährleisten.

Abbildung 216 - Das modifizierte Assemblerprogramm

Was wir jetzt noch wissen sollten, dass an Speicherstelle *$4001* der Wert zu finden ist, der über den *LDA*-Befehl an Speicherstelle *$4000* in den Akku geladen wird. Wenn wir also den Wert in Speicherstelle *$4001* von Basic heraus ändern könnten, dann ist es uns möglich, verschiedene Zeichen auf dem Bildschirm darzustellen. Das folgende BASIC-Programm wird das für uns übernehmen. Wir verlassen also mit der Eingabe von ***X*** und einer Bestätigung über die *RETURN*-Taste den Monitor und befinden uns im BASIC-Interpreter.

Abbildung 217 - Das BASIC-Programm

Gehen wir die Zeilen einmal schrittweise durch.

- **Zeile 10:** Über eine *FOR*-Schleife wird die Laufvariable *I* mit Werten zwischen *0* und *9* versorgt. Alles, was sich zwischen *FOR* und *NEXT* befindet, wird mehrfach von der Schleife abgearbeitet. Im ersten Durchlauf besitzt die Laufvariable *I* den Startwert *0*, beim zweiten *1*, beim dritten *2* und so weiter.
- **Zeile 20:** Über den *POKE*-Befehl wird ein Wert - hier der Inhalt der Laufvariablen *I* - an die genannte Speicheradresse *16385* geschrieben. Dabei handelt es sich exakt um die Adresse *$4001*, an der der Wert für den *LDA*-Befehl aus dem Maschinenprogramm zu finden ist. Der Offsetwert ist *65*, was dem Buchstaben *A* entspricht. Der Wert der Laufvariablen *I* wird diesem Offset bei jedem erneuten Schleifendurchlauf, um den Wert *1* erhöht, hinzuaddiert. Auf diese Weise werden die nachfolgenden Buchstaben kodiert.
- **Zeile 30:** Mittels *SYS 16384* wird das Maschinenprogramm über die Startadresse *$4000* aufgerufen und abgearbeitet. Dort wird der im Akku befindliche Wert über die *CHROUT*-Routine

abgearbeitet und mach der Abarbeitung die Kontrolle an Basic zurückgegeben.

- **Zeile 40:** Der *NEXT I*-Befehl leitet den nächsten Schleifendurchlauf ein, bis der Wert der Laufvariablen größer *9* ist. Es kommt demnach zu *10* Schleifendurchläufen.

Sehen wir uns das Ergebnis des Programmlaufes an.

```
RUN
ABCDEFGHIJ
READY.
```

Abbildung 218 - Das BASIC-Programm wurde ausgeführt

Die CHROUT-Routine - Teil 2

Sehen wir uns ein weiteres kleines Programm an, das diesmal Daten aus dem Speicher liest, die dort zuvor abgelegt wurden, damit die *OUTPUT*-Routine etwas Sinnvolles auf dem Bildschirm anzeigt. Wir nutzen dazu einfach einen bestimmten Bereich im Speicher, der einfach nur Werte für Daten besitzt und im eigentlichen Sinne kein ausführbares Programm darstellen. Ich nehme dazu wieder die Startadresse *$4100*. Ab dieser Adresse sollen die Buchstabenwerte für die folgende Zeichenfolge zu finden sein. Sehen wir uns zunächst die Zeichen und die entsprechenden Hex-Codes an.

Code	41	53	53	45	4D	42	4C	45	52	20
Zeichen	A	S	S	E	M	B	L	E	R	

Tabelle 36 - Die Codetabelle für Zeichenfolge - Teil 1

Code	4D	41	43	48	54	20
Zeichen	M	A	C	H	T	

Tabelle 37 - Die Codetabelle für Zeichenfolge - Teil 2

Code	53	50	41	53	53
Zeichen	S	P	A	S	S

Tabelle 38 - Die Codetabelle für Zeichenfolge - Teil 3

Um die Werte einfach in die Speicherstellen zu schreiben, wird der folgende Befehl eingegeben.

M4100 4118

```
.M4100 4118
:4100  41 53 53 45 4D 42 4C 45   ASSEMBLE
:4108  52 20 4D 41 43 48 54 20   R MACHT
:4110  53 50 41 53 53 00 FF FF   SPASS...
.
```

Abbildung 219 - Die Daten im Speicher

Und siehe da, ich habe schon die richtigen Werte eingegeben. Wie ich das schon einmal erwähnte, kann man zum Editieren der Werte einfach mit den Cursor-Tasten an die entsprechenden Stellen navigieren, den Wert ändern und dann mit der *RETURN*-Taste

bestätigen. Auf der rechten Seite ist zudem die Interpretation der Codes zu sehen und der gewünschte Text ist dort schon zu erkennen. Nachdem das also vorbereitet wurde, können wir uns dem eigentlichen Programm widmen.

```
.D4000
,4000  A9 00     LDA #00
,4002  A2 00     LDX #00
,4004  BD 00 41  LDA 4100,X
,4007  20 D2 FF  JSR FFD2
,400A  E8        INX
,400B  E0 15     CPX #15
,400D  D0 F5     BNE 4004
,400F  00        BRK
--------------------------
.
```

Abbildung 220 - Das Programm zur Anzeige der Zeichenkette

Ich denke, dass das Programm recht einfach zu verstehen ist. Über das *X*-Register werden die Speicherstellen ab Adresse *$4100* angesprochen und schrittweise erhöht. Der *CPX*-Befehl schaut nach, ob schon *21* Zeichen für die Ausgabe erreicht wurde und setzt das entsprechende Flag, was der nachfolgende *BNE*-Befehl dafür nutzt, entweder einen weiteren Schleifendurchlauf einzuleiten oder aus der Schleife auszusteigen.

Im Anschluss wird die *CHROUT*-Routine aufgerufen und das entsprechende Zeichen an der aktuellen Cursor-Position angezeigt.

```
.G4000ASSEMBLER MACHT SPASS
  PC  SR AC XR YR SP  NV-BDIZC
;4010 33 53 15 00 ED  00110011
.
```

Abbildung 221 - Die Ausgabe der Zeichenkette

Da sich die Textausgabe in derselben Zeile wie der Befehl *G4000* befindet und direkt im Anschluss ausgegeben wird, schaut das ein wenig unschön aus. Der Text sollte in der nachfolgenden Zeile starten und am Ende *2x* einen Zeilenvorschub erwirken. Wenn man in die PETSCII-Tabelle schaut, dann ist dort unter dem Code *13* (*$0D*) ***RETURN*** zu sehen, was ein *Carriage-Return* (Wagenrücklauf) bedeutet. Im eigentlichen Code müssen wir dann aufgrund der drei zusätzlichen Zeichen eine Anpassung der Abbruchbedingung für die Schleife vornehmen. Es sollen jetzt *24* statt nur *21* Zeichen ausgegeben werden.

CPX #$15

muss in

CPX #$18

geändert werden. Wir erinnern uns, dass das Editieren recht einfach geht. Einfach

D4000 oder **D400B**

eingeben, die *ESC*-Taste drücken und dann mit den Cursor-Tasten zum Wert *15* navigieren. Den Wert auf *18* ändern und mit der *RETURN*-Taste bestätigen.

```
.D4000
,4000   A9 00      LDA #00
,4002   A2 00      LDX #00
,4004   BD 00 41   LDA 4100,X
,4007   20 D2 FF   JSR FFD2
,400A   E8         INX
,400B   E0 18      CPX #18
,400D   D0 F5      BNE 4004
,400F   00         BRK
.
.
```

Abbildung 222 - Die Anpassung der Abbruchbedingung für die Schleife

So, der Programmcode wurde geändert. Im letzten Schritt müssen die Daten einer Anpassung unterzogen werden. Nachfolgend sind diese mit den zusätzlichen Werten *$0D* am Anfang und am Ende zu sehen.

```
.M4100
:4100 0D 41 53 53 45 4D 42 4C    .ASSEMBL
:4108 45 52 20 4D 41 43 48 54    ER MACHT
:4110 20 53 50 41 53 53 0D 0D     SPASS..
.
```

Abbildung 223 - Die Werte für das Carriage-RETURN wurde hinzugefügt

Wenn jetzt das Programm über die Eingabe von

G4000

gestartet wird, schaut die Ausgabe des Testes etwas übersichtlicher aus.

```
.G4000
ASSEMBLER MACHT SPASS


  PC  SR AC XR YR SP  NV-BDIZC
;4010 33 0D 18 00 DE  00110011
.
```

Abbildung 224 - Die neue Textausgabe

In BASIC kann ein Carriage-Return übrigens mit dem *CHR$(13)*-Befehl bewirkt werden.

Die GETIN-Routine

Die sogenannte *Get-Input Subroutine* besitzt den Namen *GETIN*, welche ein Byte vom Standardeingang liest. In der Regel ist das die Tastatur. Die Einsprungadresse in dieses Unterprogramm lautet *$FFE4*. Sehen wir uns dazu ein kurzes Programm an, was solange in einer Endlosschleife verharrt, bis eine Taste gedrückt wird.

```
.D4000
,4000  20 E4 FF   JSR FFE4
,4003  F0 FB      BEQ 4000
,4005  00         BRK
--------------------------
.
```

Abbildung 225 - Das Programm für die GETIN-Routine

Nach dem Start über

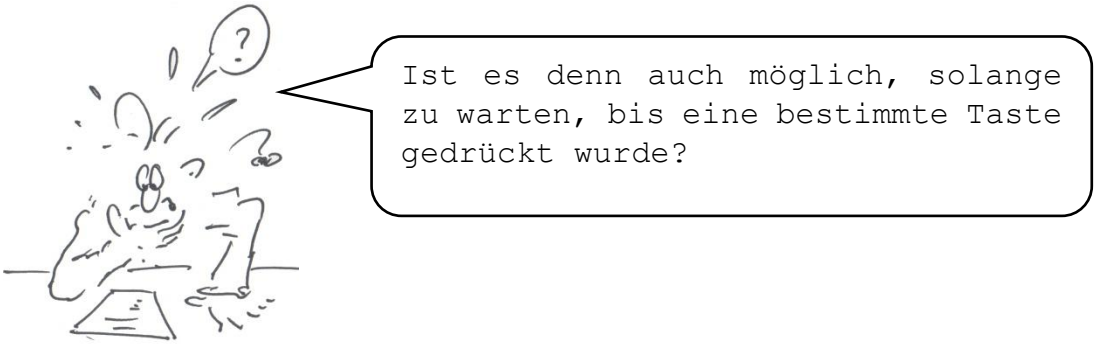
G4000

verschwindet der blinkende Cursor, weil das Programm in einer Endlosschleife hängt.

```
.G4000
```

Abbildung 226 - Nach dem Start wartet das Programm auf einen Tastendruck

Erst, wenn jetzt eine Taste gedrückt wird, wird diese Schleife verlassen und die Kontrolle an den Monitor zurückgegeben. Wie funktioniert diese *GETIN*-Routine? Es wird ja über den *BEQ*-Befehl ein Rücksprung zur Zeile bewirkt, in der ein erneuter Aufruf der *GETIN*-Routine erfolgt. Es muss sich also beim Aufruf der Routine um etwas handeln, was das *Z*-Flag beeinflusst, denn der *BEQ*-Befehl verzweigt solange, wie das *Z*-Flag gleich *1* ist.



Gute Frage! Ich mache einmal einen Test, bei dem ich verschieden Tasten drücke und dann den Inhalt des Akkus überprüfe. Dann drücke ich doch mal zuerst die Taste ***A***, starte dann das Programm erneut und drücke die Taste ***B***.

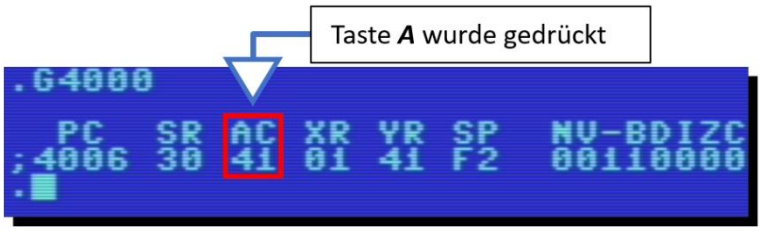


Abbildung 227 - Die Taste A wurde gedrückt

Und jetzt

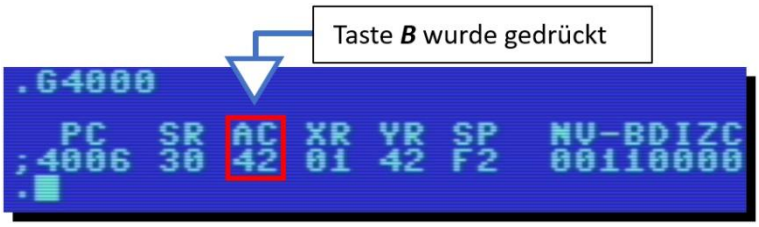


Abbildung 228 - Die Taste B wurde gedrückt

Es ist zu erkennen, dass je nach gedrückter Taste, sich ein anderer Wert im Akku befindet. Zwar habe ich das jetzt nur anhand der beiden Tasten *A* und *B* verdeutlicht, doch man kann mir glauben, dass es sich dabei um den PETSCII handelt.

Nun kann man die Funktion des Programms natürlich noch derart erweitern, dass es nur beim Drücken einer bestimmten Taste weiter im Programm geht. Das möchte ich anhand der Taste *X* zeigen. Der PETSCII-Code für die Taste *X* lautet *$58* (*88*).

```
.D4000
,4000  20 E4 FF   JSR FFE4
,4003  F0 FB      BEQ 4000
,4005  C9 58      CMP #58
,4007  D0 F7      BNE 4000
,4009  00         BRK
--------------------------
.
```

Abbildung 229 - Programm solange unterbrechen, bis die X-Taste gedrückt wird

Das Programm ist recht einfach, denn wenn eine Taste gedrückt wird, kommt es nach dem *BEQ*-Befehl zu einem erneuten Vergleich mit dem Wert *$58*, der für die *X*-Taste steht. Ist der Wert im Akku ungleich *$58*, erfolgt ein erneuter Sprung zur Adresse *$4000*, und es wird auf einen erneuten Tastendruck gewartet. Ist das dann die *X*-Taste, wird der *BRK*-Befehl in Adresse *$4009* erreicht.

Die Sache mit dem Stapel - Teil 1

Kommen wir zu einem wirklich spannenden Thema, denn hier wird einiges missverstanden und ich bin auch schon oft reingefallen. Ich hoffe, diesmal nicht. Wir haben gerade mit dem Aufruf einer System-Routine gesehen, dass diese mit einem *JSR*-Befehl aufgerufen wird. Damit der Rücksprung zum Aufrufer aus einem derartigen Unterprogramm problemlos funktioniert, muss die Rücksprungadresse irgendwo temporär zwischengespeichert werden.

Aus diesem Grund wurde ein sogenannter Stapelspeicher eingerichtet, der auch *Stack* genannt wird. Ich muss zur Erklärung des Stacks ein wenig ausholen.

Die internen Speicherbereiche der CPU – also zum Beispiel der Akkumulator oder der Programmzähler – fallen in die Kategorie *Register* und beschreiben ihre Herkunft als der CPU zugehörig und ein Teil ihrer selbst. Bei Speicherbereichen beziehungsweise –typen wie *ROM* oder *RAM* handelt es sich nicht um Register, denn sie sind der CPU über den Bus zugänglich und demnach kein unmittelbarer Teil von ihr. Beim gerade erwähnten Stack schaut es nicht anders aus. Doch wo ist dieser angesiedelt und was ist seine Aufgabe? Nun, es handelt sich dabei um einen speziellen Speicherbereich, der sich im Adressbereich von *$0100* bis *$01FF* befindet und in Summe *256* Bytes umfasst. Schön und gut und was soll das jetzt? Kommen wir zu dessen Struktur und Aufgabe. Der Stack besitzt die grundlegende Aufgabe der temporären Speicherung von Werten. Da die internen Register in der Anzahl recht knapp bemessen sind, ist der Stack im genannten Speicherbereich anzutreffen. Nicht selten kommt es in der Programmierung vor, dass wichtige Daten gespeichert werden müssen, weil kurzzeitig ein noch wichtigeres Thema bearbeitet werden muss. Nach dessen Abarbeitung beziehungsweise Beendigung kann es dann an vorheriger Stelle mit den gesicherten Daten weitergehen. Was könnte das für ein Szenario sein, auf das ich hier abziele? Ganz einfach! Es geht um den Aufruf einer Unterroutine aus dem Hauptprogramm heraus, wie wir das gerade eben gemacht haben. Wurde dieses Unterprogramm abgearbeitet, muss die Ausführung des Hauptprogramms natürlich an geeigneter Stelle fortgesetzt werden, als wenn das Unterprogramm überhaupt nicht existiert hätte.

Was ist ein Unterprogramm?

Werden in einem Computerprogramm an mehreren Stellen immer wieder die gleiche Codesegmente benötigt, ist es sinnvoll, diese in ein sogenanntes *Unterprogramm* in Form einer Funktion auszulagern. Nun kann von unterschiedlichsten Stellen im Computerprogramm ein Aufruf zu diesem Unterprogramm erfolgen. Nach der Abarbeitung des Unterprogramms verzweigt die Programmausführung zurück an die aufrufende Stelle und setzt dort das eigentliche Hauptprogramm fort.

Sehen wir uns dazu die folgende Abbildung an.

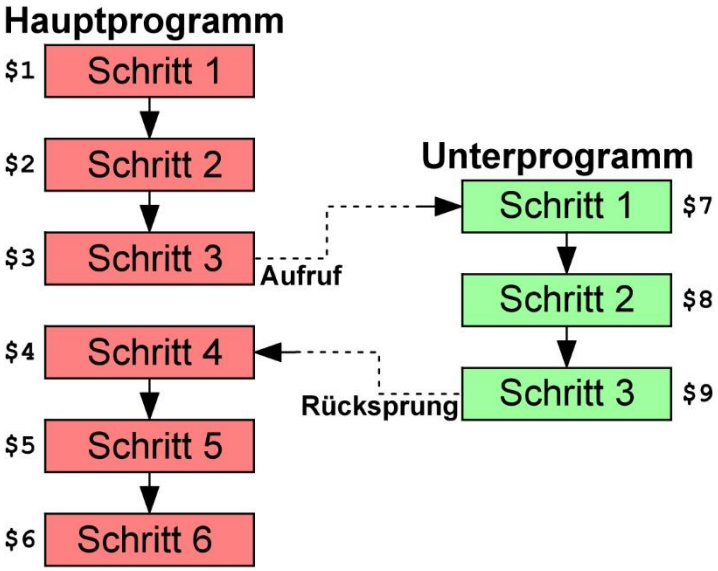


Abbildung 230 - Hauptprogramm ruft Unterprogramm

Auf der linken Seite ist ein Hauptprogramm mit 6 Schritten zu sehen, dass sich im fiktiven Adressbereich von *$1* bis *$6* befindet. Auf der rechten Seite ist ein Unterprogramm mit drei Schritten zu sehen, das sich im fiktiven Adressbereich von *$7* bis *$9* befindet. Würde das Hauptprogramm keinen Unterprogrammaufruf besitzen, wäre die Sache mit dem Programmzähler recht einfach. Der Aufruf der Adressen *$1* bis *$6* bestünde im simplen Hochzählen der Adresse *$1* bis zum Ende des Programms an Adresse *$6*. Nun gibt es aber das Unterprogramm, das vom Hauptprogramm an Adresse *$3* aufgerufen wird. Es erfolgt ein Sprung zur Adresse *$7* und der Programmzähler arbeitet in bekannter Weise dort wieder die Adressen ab. Nach Beendigung des Unterprogramms an Adresse *$9* soll es jetzt zurück zum Hauptprogramm gehen. Doch hoppla! An welcher Stelle geht es denn jetzt weiter??? Klar, wir sehen das hier auf der Abbildung recht eindeutig, doch die CPU hat keine Ahnung, an welcher Stelle denn nun der Aufruf zum Unterprogramm erfolgte und wo es dann später weitergehen soll! Die Lösung des Problems besteht in der Sicherung der sogenannten Rücksprungadresse vor der Abarbeitung des Unterprogramms. Diese Rücksprungadresse ist hier *$4*, die bei Schritt 4 des Hauptprogramms angesiedelt ist. Hier kommt nun der Stack ins Spiel. Es gibt bei der *6502/6510*-CPU zwei Befehle, die quasi Hand-In-Hand arbeiten. Das sind

- *JSR* (Jump to Subroutine) – Springe zum Unterprogramm
- *RTS* (Return from Subroutine) – Rücksprung vom Unterprogramm

Der Befehl *RTS* weiß eigentlich nicht, an welcher Stelle denn nun zurückgesprungen werden soll, denn er besitzt keinen Parameter in Form einer Adresse. Doch er nutzt dazu den Stack, auf dem

diese Adresse zu finden ist, denn der Befehl *JSR* hat diese Rücksprungadresse dort abgelegt. Da ein Unterprogramm aus einem Hauptprogramm natürlich an mehreren Stellen aufgerufen werden kann, ist die Rücksprungadresse auch jedes Mal eine andere. Diese Flexibilität bietet die Struktur des Stacks, denn es besteht ja die Möglichkeit, dass ein Unterprogramm ein anderes Unterprogramm aufrufen will und im Endeffekt erfolgen dann mehrere Rücksprünge aus verschiedenen Unterprogrammen, die alle einzeln abgearbeitet werden müssen, bis es letztendlich wieder mit dem Hauptprogramm weitergeht, wie das auf der folgenden Abbildung zu erkennen ist.

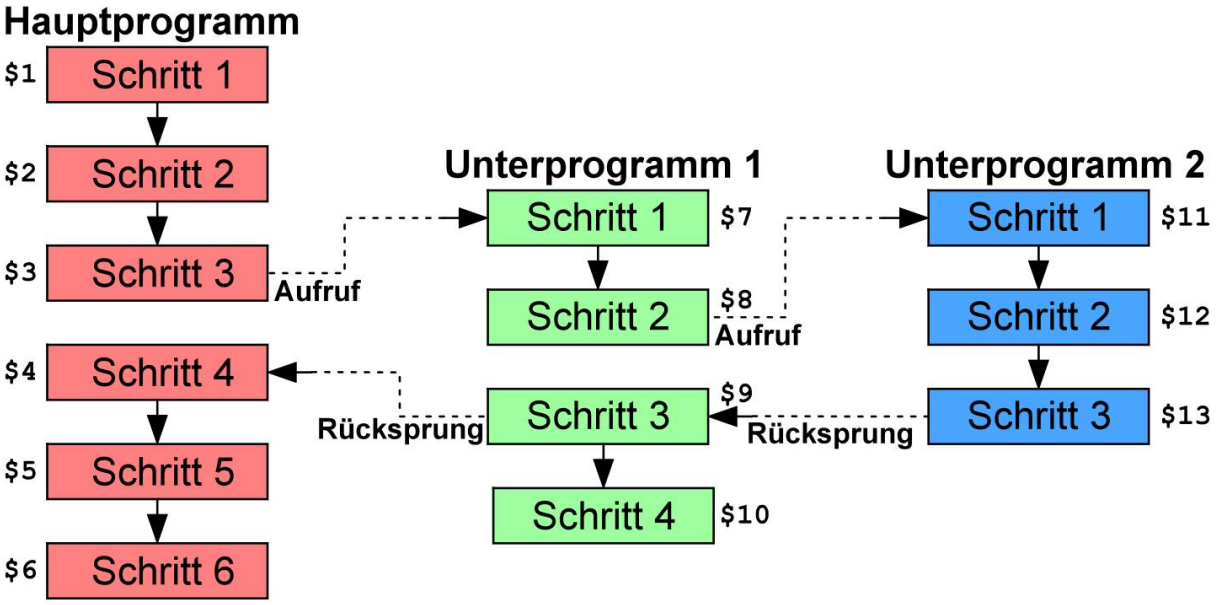


Abbildung 231 - Mehrfacher Aufruf eines Unterprogramms

Bei diesem Beispiel müssen also zwei Rücksprungadressen *$4* und *$9* gesichert werden und natürlich in der richtigen Reihenfolge abgearbeitet werden. Aus diesem Grund ist der Stack wie folgt organisiert.

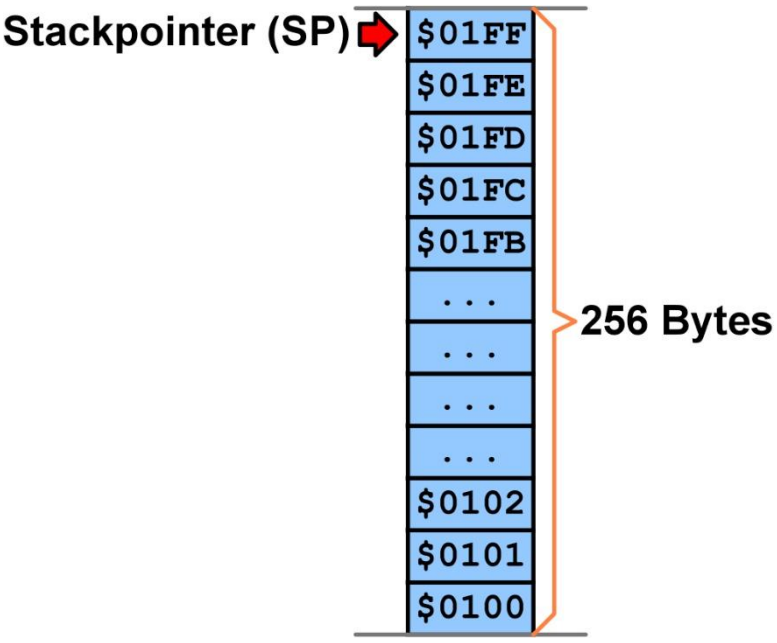


Abbildung 232 - Der Stack im Speicher

Im Adressbereich von *$0100* bis *$01FF*, der im RAM angesiedelt ist, befinden sich diese reservierten Speicherstellen, die von der CPU exklusiv als Stack-Bereich genutzt werden. Sie sollten auf keinen Fall durch ein Programm in irgendeiner Weise direkt und ohne Grund beschrieben werden, denn das hätte unweigerlich einen Absturz des Programms zur Folge. Der sogenannte *Stackpointer*, also der *Stapelzeiger* in Form eines *8*-Bit breiten Registers weist immer auf das nächste freie Byte des Stapels. Ganz zu Beginn ist dies in der Regel die Adresse *$01FF*. Diesem Register mit der Breite von *8* Bits ist jedoch ein neuntes Bit auf der linken Seite zugeordnet. Kommen wir noch mal zurück zum Stapel, der nach seiner eigentlichen Funktion die sogenannte *LIFO*-Struktur (last-in-first-out) vorweist. Übersetzt bedeutet das: Das zuletzt abgelegte Element wird auch wieder zuerst ausgegeben. Um sich das bildlich vorzustellen, kann ein Aktenstapel bemüht werden, auf dem nach und nach neue Hefter oben abgelegt werden. Will man eine Akte bearbeiten, wird diese wieder von oben heruntergenommen. Die zuletzt abgelegte kommt also zuerst in Betracht. Das erste Stapelelement – hier das dunkelblaue Element - befindet sich naturgemäß auf dem Boden des Stapels, das letzte Stapelelement – hier das Gelbe - auf dessen Spitze. Um also an die zuvor abgelegten Elemente des Stapels zu gelangen, muss dieser von oben nach unten quasi abgetragen werden.

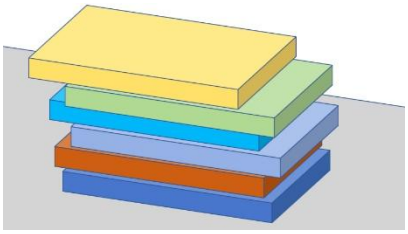


Abbildung 233 - Irgendein Stapel

Sehen wir uns dazu einmal ein Beispiel an, bei dem zwei Befehle genutzt werden, um einerseits etwas auf den Stapel abzulegen, was *Push* genannt wird und andererseits etwas vom Stapel zu nehmen, was *Pull* oder *Pop* genannt wird.

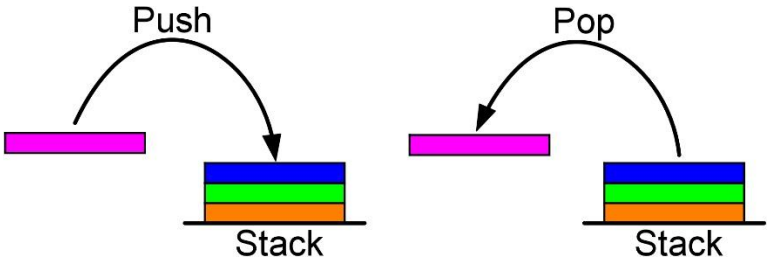


Abbildung 234 - Stack-Aktionen

Zu sehen ist hier zu Beginn die Ausgangssituation, wo sich der SP an der initialen Adresse *$01FF* befindet. Wir betrachten den Stack aus der entgegengesetzten perspektive, der quasi auf den Kopf gestellt ist und von oben nach unten wächst. Die genannten Befehle *Push* und *Pop* sind manuelle Aktionen, die den Stack beeinflussen. Ich komme noch darauf zurück, doch im Moment möchte ich die Funktion des Stacks mit dem *JSR*- und *RTS*-Befehl erläutern, die das automatisch machen.

Sehen wir uns das Ganze an einem konkreten Beispiel an, was zwar im eigentlichen Sinn keine brauchbare Funktion besitzt, doch das Zusammenspiel von Adressen, Stapel-Zeiger (Stack-Pointer) und Stack selbst recht gut zeigt. Nach dem Start des Computers zeigt der Stack-Pointer auf eine bestimmte Adresse, die sich meist am oberen Ende des entsprechenden Speichers befindet. Hier weist er auf die Adresse *$F7*, was in der Realität die Adresse *$01FF* ist.

Abbildung 235 - Der Stack-Pointer

Der Stack-Pointer weist also auf die genannte Adresse, was wie folgt ausschaut.

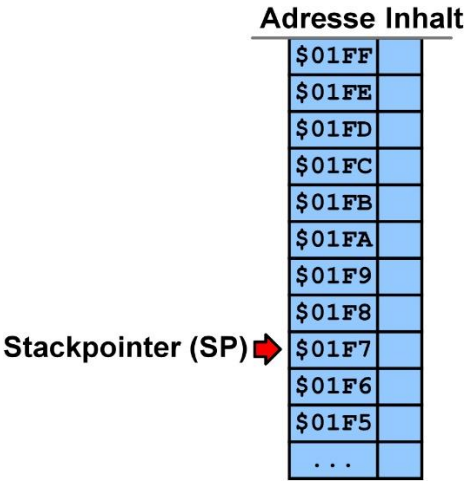


Abbildung 236 - Der Stackpointer weist auf Adresse $01F7

Soweit so gut und sehen wir uns jetzt das kleine Programm an.

```
.D4000
,4000   A9 02      LDA #02
,4002   18         CLC
,4003   69 06      ADC #06
,4005   8D 50 40   STA 4050
,4008   18         CLC
,4009   20 00 41   JSR 4100
,400C   00         BRK
--------------------------------
.
```

Abbildung 237 - Das Hauptprogramm

Es werden einige belanglose Befehle ausgeführt und ab Speicherstelle *$4009* erfolgt der Aufruf des Unterprogramms über den *JSR*-Befehl. Dieses Unterprogramm besteht lediglich aus einem *RTS*-Befehl und soll den Rücksprung zum Hauptprogramm an die korrekte Adresse ermöglichen.

```
.D4100
,4100   60         RTS
--------------------------------
.
```

Abbildung 238 - Das Unterprogramm

Wer in Basic den *GOSUB*-Befehl verwendet hat, kennt die Funktionsweise des abschließenden *RETURN*-Befehls, der die *GOSUB*-Anweisung respektive das Unterprogramm abschließt. Ganz ähnlich läuft es in einem Maschinenprogramm ab. Man könnte also folgendes formulieren.

Basic	**Assembler**
GOSUB	entspricht *JSR*
RETURN	entspricht *RTS*

Starten wir jetzt das Hauptprogramm über den schon bekannten *Trace-Walk.*

TW4000

Für die folgende Anzeige bzw. den Programmablauf wurde von mir 4x die Leertaste gedrückt.

Abbildung 239 - Das schrittwiese Abarbeiten der Programmzeilen

Der nächste auszuführende Befehl bei einem Druck auf die Leertaste wäre dann der *JSR $4100*-Befehl, was einen Sprung zur genannten Speicheradresse bewirkt. Es ist zu sehen, dass sich der Programm-Zähler nun auf der Adresse *$4100* befindet, weil der Sprung genau dorthin erfolgen sollte.

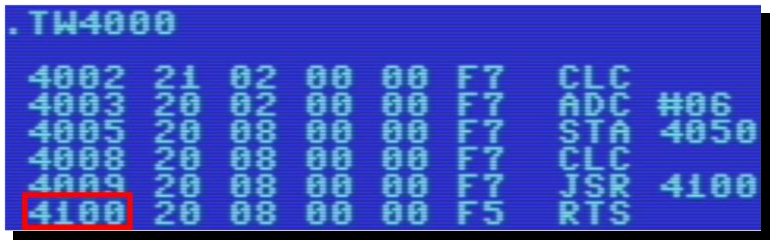


Abbildung 240 - Der Sprung ins Unterprogramm ist erfolgt

Nun können wir den Trace-Walk erst einmal mit dem Druck auf die *ESC*-Taste in VICE beenden. Wir wollen und den Inhalt des Stacks ansehen und was dort für Werte zu sehen sind. Das machen wir mit dem ***M***-Kommando von SMON, was den Speicher (Memory) anzeigt. Ich habe die folgende Startadresse eingegeben.

M01F6

Sehen wir uns die Ausgabe des Befehls zuerst einmal an.

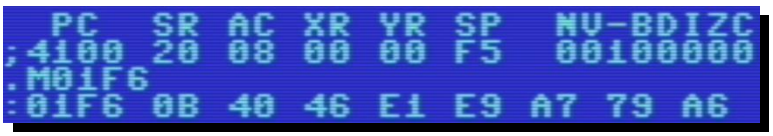


Abbildung 241 - Die Anzeige der Speicherinhalte

Ich denke mal, dass das hier auf den ersten Blick nichtssagend ist und dennoch ist hier die Rücksprungadresse verborgen. Zwar nicht ganz, denn es muss dazu noch der Wert *1* zu einer Speicherstelle hinzuaddiert werden, doch es ist hier alles korrekt. Gehen wir wieder ein paar Schritte zurück und sehen uns noch einmal das Hauptprogramm etwas genauer an. Für uns wichtig ist die markierte Zeile mit dem Aufruf des Unterprogramms.

Abbildung 242 - Das Hauptprogramm

Die entsprechenden Speicherstellen sind

- *$4009* : *$20*
- *$400A* : *$00*
- *$400B* : *$41*

Hier muss irgendwie verborgen sein, was auf den Stack gewandert ist. Es ist hier wichtig zu wissen, dass die Werte ***$400B*** auf den Stack geschoben wurden und nicht die eigentliche Adresse *$400C*, wo es nach der Abarbeitung des Unterprogramms eigentlich weitergehen soll. Der Aufruf des *JSR*-Befehls schiebt dabei zuerst das High-Byte (***$40***) und dann das Low-Byte (***$0B***) nacheinander auf den Stack.

Konkret bedeutet das:

- Der Stackpointer weist auf die Speicherastelle *$01F6*
- Die Ablage des High-Bytes *$40* an diese Adresse
- Den Stackpointer um den Wert *1* vermindern. Er weist jetzt auf die Adresse *$01F5*
- Die Ablage des Low-Bytes *$0B* an diese Adresse
- Den Stackpointer um den Wert *1* vermindern. Er weist jetzt auf die Adresse *$01F4*

Wenn der *RTS*-Befehl des Unterprogramms aufgerufen wird, werden Low- und High-Byte wieder vom Stack in umgekehrter Reihenfolge genommen, was zur Adresse *$400B* führt und die CPU addiert dann den Wert *$01* hinzu, was zu *$400C* als Rücksprungadresse führt.

Stimmt das auch mit der Speicheranzeige überein, die wie eben gesehen haben? Sehen wir nach.

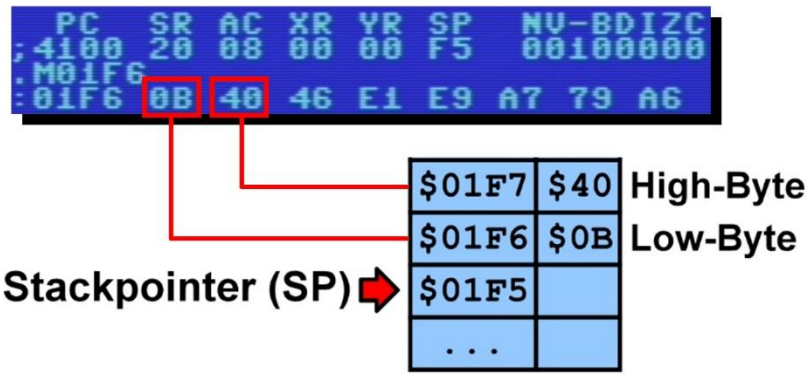


Abbildung 243 - Der Stack und die Rücksprungadresse

Das passt also. Es ist zudem zu erkennen, dass der Stack-Pointer (SP) auf die Adresse *$F5*, also *$01F5* weist. Nun wollen wir aber das Programm inklusive des abgearbeiteten *RTS*-Befehls des Unterprogramms analysieren. Was passiert dann mit dem Stack-Pointer? Da wir den Programmablauf mit dem Drücken der *ESC*-Taste unterbrochen haben, würde ein Neustart des Trace-Walks mit der Eingabe von

TW4000

den Stack-Pointer nicht auf seinen vorherigen Startwert *$F7* setzen, sondern mit der aktuellen Position *$F5* starten. Das wollen wir nicht. Man kann - ohne, dass das Programm im Speicher gelöscht wird - einen CPU-Reset über den Menüpunkt ***File > Reset > Reset machine CPU*** durchführen.

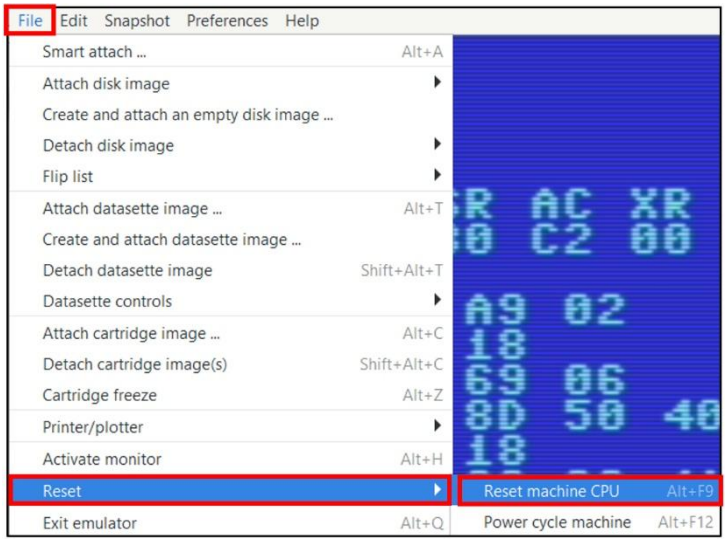


Abbildung 244 - Ein CPU-Reset in VICE

Jetzt gelangen wir über die Eingabe von

SYS 49152

erneut in den SMON und geben erneut den Befehl für das Starten des Trace-Walks ein.

TW4000

Nun drücken wir *7x* die Leertaste, so dass der *BRK*-Befehl des Unterprogramms abgearbeitet und Trace-Walk beendet wurde. Der Cursor blinkt wieder und zeigt den Inhalt der Register an. Und was sehen wir dort? Der Stack-Pointer ist wieder nach oben zur vorherigen Startadresse *$F7* gewandert, weil die beiden *8*-Bit-Adressen für den Rücksprung ins Hauptprogramm vom Stack heruntergenommen wurden.

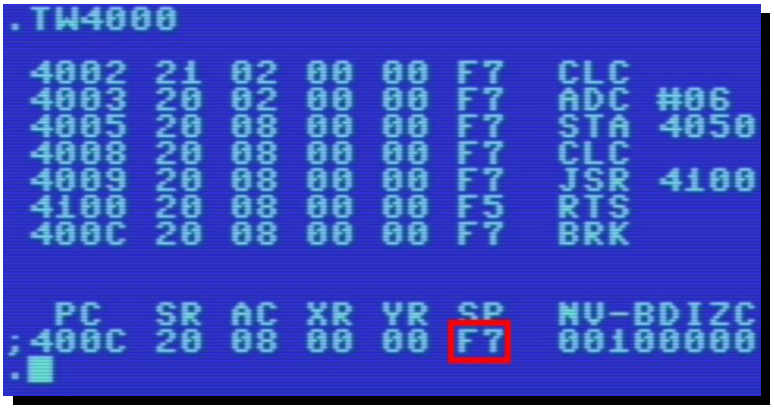


Abbildung 245 - Der Stack-Pointer besitzt seinen vorherigen Wert $F7

Zudem können wir erkennen, dass der Programmzähler von *$4100* auf *$400C* gesprungen ist und wieder in den Bereich des Hauptprogramms weist.

Die Sache mit dem Stapel - Teil 2

Kommen wir zu einem weiteren Beispiel, das sich mit dem Stack befasst. Im ersten Teil hatte ich die grundlegende Funktionalität erläutert und wie es manuell möglich ist, über Push etwas auf den Stapel abzulegen und mit Pop etwa herunterzunehmen. Hierbei handelt es sich um manuelle Aktionen, die über entsprechende Assembler-Befehle auf den Stack einwirken. Zuvor haben die Befehle *JSR* und *RTS* das quasi im Hintergrund erledigt. Nun machen wir das einmal selbst. Sehen wir uns zuvor einmal die entsprechenden OP-Codes an, die die Aufgaben übernehmen.

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r •	6
7	BVS r	ADC (zp),y	ADC (zp) *		STZ zp,x •	ADC zp,x	ROR zp,x	RMB7 zp •	SEI i	ADC a,y	PLY s •		JMP (a,x) *	ADC a,x	ROR a,x	BBR7 r •	7

Abbildung 246 - Die OP-Code-Matrixtabelle für die Op-Codes PHA und PLA

Der PHA-Befehl:

Der *PHA*-Befehl (***P**us**H** **A***) besitzt den OP-Code *$48* und wir sehen unterhalb des Mnemonics ein kleines ***s*** stehen. Was hat es nun wieder damit auf sich? Sehen wir dazu auf Seite *20* des W65C02 Dokumentes nach und finden dort in der *Addressing Mode Table* folgenden Hinweis.

Address Mode	Instruction Times in Memory Cycle		Memory Utilization in Number of Program Sequence Bytes	
	Original NMOS 6502	W65C02S	Original NMOS 6502	W65C02S
1. Absolute a	4 (3)	4 (3)	3	3
2. Absolute Indexed Indirect (a,x)	-	6	-	3
3. Absolute Indexed with X a,x	4 (1,3)	4 (1,3)	3	3
4. Absolute Indexed with Y a,y	4 (1)	4 (1)	3	3
5. Absolute Indirect (a)	5	6	3	3
6. Accumulator A	2	2	1	1
7. Immediate #	2	2	2	2
8. Implied i	2	2	1	1
9. Program Counter Relative r	2 (1,2)	2 (1,2)	1	1
10. Stack s	3-7	3-7	1	1
11. Zero Page zp	3 (3)	3 (3)	2	2

Abbildung 247 - Der Address-Mode für den Stack

Aha, es handelt sich also um eine Operation, die sich auf den Stack bezieht. Wer hätte es gedacht?! Der PHA-Befehl beeinflusst keine Flags.

N	V	-	B	D	I	Z	C
-	-	-	-	-	-	-	-

Tabelle 39 - Die Flag-Beeinflussung des PHA-Befehls

Operation: A↓ (Dieser Befehl überträgt den aktuellen Wert des Akkumulators an die nächste freie Stelle auf dem Stack, wobei der Stack-Pointer automatisch dekrementiert wird und auf die nächste leere Stelle zeigt.)

Der PLA-Befehl:

Der *PLA*-Befehl (***P**u**L**l** **A***) besitzt den OP-Code *$68* und wir sehen unterhalb des Mnemonics ebenfalls ein kleines ***s*** stehen.

Der *PLA*-Befehl beeinflusst im Gegensatz zum *PHA*-Befehl wohl einige Flags, denn der Wert vom aktuellen Stack-Pointer wird in den Akku geladen.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 40 - Die Flag-Beeinflussung des PLA-Befehls

Operation: A↑ (Dieser Befehl lädt den Inhalt des Wertes vom Stack, auf den der aktuelle Stack-Pointer weist in den Akku. Im Anschluss wird der Stack-Pointer inkrementiert.)

Sehe wir uns dazu ein kleines Programm an, das den Wert des Akkus zur Sicherung auf den Stack legt, um ihn im Anschluss von dort wieder zu laden.

```
  PC   SR AC XR YR SP   NV-BDIZC
;4103 B0 98 00 00 F7   10110000
.D4000
,4000   A9 A1      LDA #A1
,4002   48         PHA
,4003   A9 00      LDA #00
,4005   68         PLA
,4006   00         BRK
--------------------------------
.
```

Abbildung 248 - Der Akku wird auf den Stack gesichert und wieder gelesen

Es ist zu sehen, dass der Akku zu Beginn mit dem Wert *#$A1* geladen wird. Im nächsten Schritt kommt es zur Sicherung des Akkus durch den *PHA*-Befehl auf den Stack. Anschließend wird der Inhalt des Akkus über *LDA #$00* gelöscht. Letztendlich holen wir dann über den *PLA*-Befehl den zuvor auf den Stack gelegten Wert zurück in den Akku. Es ist zu sehen, dass der Stack-Pointer auf die Adresse *$F7* weist. Sehen wir nach, was der Start des Programms über

G4000

als Ergebnis liefert.

```
.G4000

  PC   SR AC XR YR SP   NV-BDIZC
;4008 B1 A1 00 00 F7   10110001
.
```

Abbildung 249 - Der Akku wieder hergestellt

Es ist zu sehen, dass sich der zuvor gesicherte Akku-Inhalt wieder auf dem Ursprungswert *$A1* befindet und der Stackpointer ebenfalls den ursprünglichen Wert *$F7* angenommen hat. Doch sehen wir uns das noch mal im Detail an. Die Ausgangssituation ist die folgende.

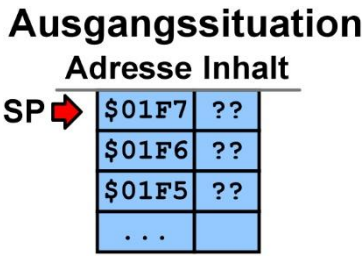


Abbildung 250 - Der Stack-Pointer weist auf die Adresse $01F7

Die Fragezeichen bedeuten irgendeinen Wert, der sich dort befindet und nicht weiter von Bedeutung ist.

Im ersten Schritt wird der Akku-Inhalt über einen *Push* auf den Stack gesichert. Der Stack-Pointer wird inkrementiert.

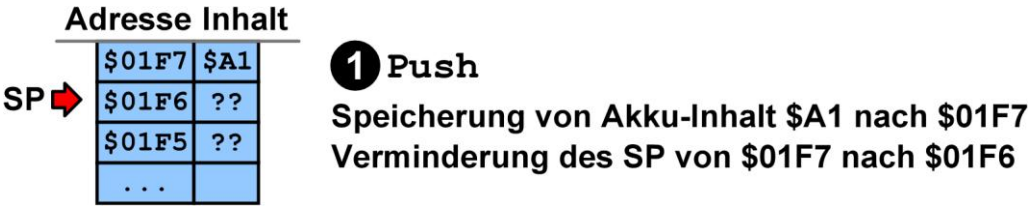


Abbildung 251 - Der Akku wird auf den Stack gesichert

Im nächsten Schritt wird der Akku über LDA #$00 gelöscht. Anschließend kommt es zum Abrufen des zuvor gesicherten Akku-Inhaltes vom Stack in den Akku. Der Stack-Pointer wird dekrementiert.

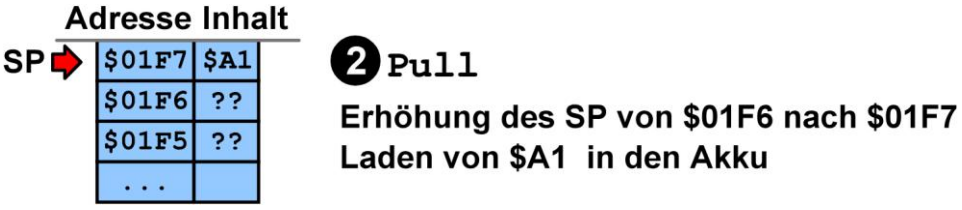


Abbildung 252 - Der Akku wird vom Stack geholt

Interessant wird es natürlich, wenn mehrere Werte auf den Stack geschoben werden. Da muss man dann wirklich nach dem Prinzip *Last In First Out* vorgehen. Dann sehen wir uns ein Beispiel an, bei dem sowohl der Inhalt des Akkus, als auch der des *X*-Registers auf den Stack wandern, um die Werte später von dort wieder zurückzuholen. Doch da werden wir mit einem Problem konfrontiert, denn es gibt keinen Befehl, der den Inhalt des *X*-Registers auf dem Stack ablegt. Beim *Y*-Register sieht es nicht anders aus. Ok, dann ist das Vorhaben wohl nicht zu realisieren?! Blödsinn! Wir müssen einen Zwischenschritt einlegen, und alles über den Akku abwickeln. Also zuerst den Inhalt des Akkus auf den Stack legen, dann den Inhalt des X-Registers in den Akku transferieren und dann wiederum den Inhalt des Akkus auf den Stack schieben. Von hinten durch die Brust ins Auge also.

Sehen wir uns dazu das folgende Programm an, was ein wenig länger als die Bisherigen ist. Zudem kommen zwei neue Befehle hinzu. Doch keine Panik!

```
  PC   SR AC XR YR SP   NV-BDIZC
;C00B  B0 C2 00 00 F7   10110000
.D4000
,4000   A9 17      LDA #17
,4002   A2 09      LDX #09
,4004   48         PHA
,4005   8A         TXA
,4006   48         PHA
,4007   A9 00      LDA #00
,4009   A2 00      LDX #00
,400B   68         PLA
,400C   AA         TAX
,400D   68         PLA
,400E   00         BRK
--------------------------------
.
```

Abbildung 253 - Der Akku und das X-Register wandern auf den Stack

Ich denke, ich zerlege das Programm in kleine Häppchen und erläutere die entsprechenden Bereiche. Leider sind in SMON, da es sich um ein Monitor-Programm und keinen richtigen Assembler handelt, keine Kommentarzeilen möglich, da alles direkt im Speicher passiert. Doch wir lassen uns dadurch nicht entmutigen.

Schritt 1: Akku und X-Register werden mit Werten initialisiert:

```
,4000   A9 17      LDA #17
,4002   A2 09      LDX #09
```

Schritt 2: Der Inhalt des Akku wird auf dem Stack abgelegt:

```
,4004   48         PHA
```

Schritt 3: Der Inhalt X-Registers wird in den Akku transferiert:

```
,4005   8A         TXA
```

Aha, hier ist ein neuer Befehl zu sehen. Der *TXA*-Befehl transferiert den Inhalt des *X*-Registers in den Akku. Es heißt hier nicht *Laden*, sondern *transferieren*, denn der Wert, mit dem gearbeitet werden soll, befindet sich schon in einem Register und soll in ein anderes übertragen, also transferiert werden.

Nachfolgend die Matrix für den *TXA*-Befehl.

OP-Code: 8A

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2
3	BMI r	AND (zp),y	AND (zp) *		BIT zp,x *	AND zp,x	ROL zp,x	RMB3 zp •	SEC i	AND a,y	DEC A *		BIT a,x *	AND a,x	ROL a,x	BBR3 r •	3
4	RTI s	EOR (zp,x)				EOR zp	LSR zp	RMB4 zp •	PHA s	EOR #	LSR A		JMP a	EOR a	LSR a	BBR4 r •	4
5	BVC r	EOR (zp),y	EOR (zp) *			EOR zp,x	LSR zp,x	RMB5 zp •	CLI i	EOR a,y	PHY s •			EOR a,x	LSR a,x	BBR5 r •	5
6	RTS s	ADC (zp,x)			STZ zp •	ADC zp	ROR zp	RMB6 zp •	PLA s	ADC #	ROR A		JMP (a)	ADC a	ROR a	BBR6 r •	6
7	BVS r	ADC (zp),y	ADC (zp) *		STZ zp,x •	ADC zp,x	ROR zp,x	RMB7 zp •	SEI i	ADC a,y	PLY s •		JMP (a,x) *	ADC a,x	ROR a,x	BBR7 r •	7
8	BRA r •	STA (zp,x)			STY zp	STA zp	STX zp	SMB0 zp •	DEY i	BIT # *	TXA i		STY a	STA a	STX a	BBS0 r •	8

Abbildung 254 - Die OP-Code-Matrixtabelle für den Op-Code TXA

Das ***i*** unterhalb des *TXA*-Befehls bedeutet, dass es sich um eine implizite Adressierung handelt.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 41 - Die Flag-Beeinflussung des TXA-Befehls

Operation: : X → A (Der TXA-Befehl verschiebt den Wert, der sich im X-Register befindet, in den Akkumulator, ohne den Inhalt des X-Registers zu verändern.)

Da sich jetzt eine Kopie des X-Registers im Akku befindet, kann der Inhalt des Akkus auf den Stack geschoben werden. Über diesen Umweg ist das genannte Problem des fehlenden Befehls behoben.

Schritt 4: Der Inhalt des Akku wird auf dem Stack abgelegt:

```
.4006  48        PHA
```

Somit sind beide Werte auf dem Stack gelandet. Im nächsten Schritt werden die Werte des Akkus und des *X*-Registers auf *0* gesetzt, um später zu sehen, ob die gespeicherten Werte korrekt vom Stack in die entsprechenden Register übertragen werden.

Schritt 5: Akku und X-Register auf 0 setzen:

```
.4007  A9 00     LDA #00
.4009  A2 00     LDX #00
```

Schritt 6: Den zuletzt abgelegten Wert vom Stack holen:

```
.400B  68        PLA
```

Doch Vorsicht, es handelt sich hier um den zuletzt auf den Stack gelegten Wert, der aus dem *X*-Register stammte und den Umweg über

den Akku nehmen musste. Dieser muss jetzt vom Akku in das *X*-Register transferiert werden.

Schritt 7: Den Inhalt des Akku in das X-Register transferieren:

```
,400C  AA          TAX
```

Und hier kommt wieder ein neuer Befehl. Der *TAX*-Befehl transferiert den Inhalt des Akkus in das *X*-Register. Es heißt hier wiederum nicht Laden, sondern transferieren.

Nachfolgend die Matrix für den *TAX*-Befehl.

OP-Code: AA

A	LDY #	LDA (zp,x)	LDX #		LDY zp	LDA zp	LDX zp	SMB2 zp •	TAY i	LDA #	TAX i		LDY A	LDA a	LDX a	BBS2 r •	A
B	BCS r	LDA (zp),y	LDA (zp) •		LDY zp,x	LDA zp,x	LDX zp,y	SMB3 zp •	CLV i	LDA A,y	TSX		LDY a,x	LDA a,x	LDX a,y	BBS3 r •	B
C	CPY #	CMP (zp,x)			CPY zp	CMP zp	DEC zp	SMB4 zp •	INY i	CMP #	DEX	WAI i •	CPY a	CMP a	DEC a	BBS4 r •	C
D	BNE r	CMP (zp),y	CMP (zp) •			CMP zp,x	DEC zp,x	SMB5 zp •	CLD i	CMP a,y	PHX s •	STP i •		CMP a,x	DEC a,x	BBS5 r •	D
E	CPX #	SBC (zp,x)			CPX zp	SBC zp	INC zp	SMB6 zp •	INX i	SBC #	NOP		CPX a	SBC a	INC a	BBS6 r •	E
F	BEQ r	SBC (zp),y	SBC (zp) •			SBC zp,x	INC zp,x	SMB7 zp •	SED i	SBC a,y	PLX s •			SBC a,x	INC a,x	BBS7 r •	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

Abbildung 255 - Die OP-Code-Matrixtabelle für den Op-Code TAX

Das ***i*** unterhalb des *TAX*-Befehls bedeutet, dass es sich um eine implizite Adressierung handelt.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 42 - Die Flag-Beeinflussung des TAX-Befehls

Operation: : A → X (Der *TXA*-Befehl verschiebt den Wert, der sich im Akkumulator befindet, in das *X*-Register ohne den Inhalt des Akkus zu verändern.)

Schritt 8: Den Inhalt des Akku vom Stack holen:

Nun kann der ursprüngliche Wert des Akkus vom Stack geholt werden.

```
,400D  68          PLA
```

Schritt 9: Das Programm wird unterbrochen:

```
,400E  00          BRK
```

Sehen wir uns jetzt die Register nach dem Start des Programms durch

G4000

an. Es ist zu sehen, dass sich die ursprünglichen Werte in den Registern befinden.

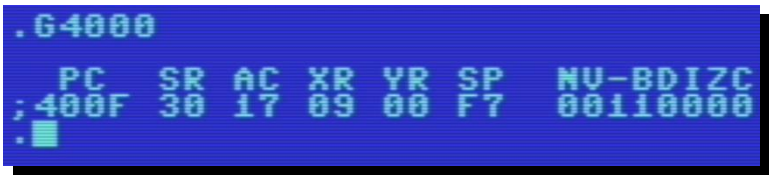


Abbildung 256 - Die Registerinhalte sind korrekt restauriert worden

Hier einige abschließende Worte zur Stack-Behandlung. Erfolgen durch den *PHA*-Befehl Werteablagen auf dem Stack, so müssen diese später auch in der gleichen Anzahl durch *PLA*-Befehle wieder vom Stack genommen werden. *PHA* und *PLA* bilden immer ein Pärchen. Kommt es hier zu einem Ungleichgewicht, dann sind Probleme unvermeidlich. Ebenso schaut es mit der Verwendung von *JSR*-Befehlen und *RTS*-Befehlen aus, die gleichfalls zusammengehören. Fehlt einer Unterroutine der abschließende *RTS*-Befehl, läuft das Programm hier einfach weiter und erledigt sicherlich nicht die gewünschte Aufgabe. Wenn ich hier von Ablegen von Werten auf dem Stack und Werte vom Stack nehmen spreche, hat das im Grunde genommen etwas mit der Positionierung des Stack-Pointers zu tun. Einen Wert vom Stack nehmen bedeutet ja das Lesen an der aktuellen Stack-Pointer-Position und der darauffolgenden Inkrementierung des Stack-Pointers. Der Wert verbleibt auf dem Stack, bis er später bei einem erneuten *PHA*-Befehl überschrieben wird.

Einige wichtige Zeiger

Einige wichtige Zeiger

Bestimmte Bereich im Speicher sind nicht immer in Stein gemeißelt, sondern können variabel sein. Damit dies auch transparent ist, gibt es Pointer, die auch Zeiger genannt werden, Sie weisen auf die Startadresse des jeweiligen Bereiches. Da wir es jedoch mit einem *8*-Bit breiten Datenbus zu tun haben, könnte ein Zeiger nur einen Bereich von *0* bis *255* abdecken. Nun wissen wir schon, dass es ein Low- und ein High-Byte gibt, um den kompletten Speicher von *64K* abzudecken. Ein Pointer zeigt immer auf das Low-Byte, wobei das High-Byte eine Adresse höher angesiedelt ist.

Auf der folgenden Abbildung sind wichtige Zeiger zu sehen.

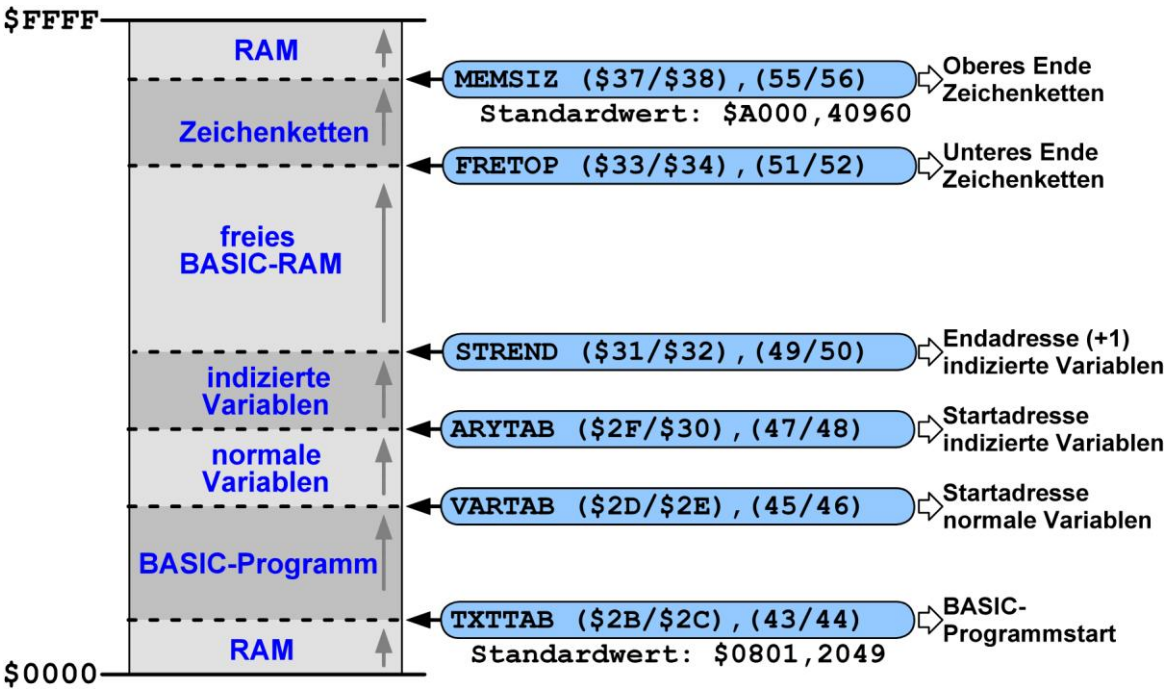


Abbildung 257 - Wichtige Zeiger

Wir können also folgendes hinsichtlich bestimmter Speicheradresse zusammenfassend sagen:

Bereich	Startadresse
Beginn von BASIC	PEEK(43) + 256·PEEK(44)
Beginn der einfachen Variablen	PEEK(45) + 256·PEEK(46)
Beginn der Arrays	PEEK(47) + 256·PEEK(48)
Zeichenketten unteres Ende	PEEK(51) + 256·PEEK(52)
Zeichenketten oberes Ende	PEEK(55) + 256·PEEK(56)

Tabelle 43 - Zusammenfassung einiger wichtiger Startadressen

Nachfolgend werden wir uns ein paar dieser Zeiger ansehen, was unter anderem wichtig ist, wenn es um die Speicherung von Daten geht. Diese Zeiger - besser Pointer genannt - müssen variabel sein, da die Daten im Anschluss des BASIC-Programms angesiedelt sind. Je nach Länge des Programms wandern die Pointer entsprechend nach oben.

Ein BASIC-Programm im Speicher

Ein BASIC-Programm im Speicher

Wenn ein Programm in BASIC geschrieben wurde, so muss dieses natürlich auch irgendwo im Speicher angelegt sein. Natürlich nicht irgendwo, sondern an einer bestimmten Stelle, die standardmäßig den folgenden Bereich vorweist.

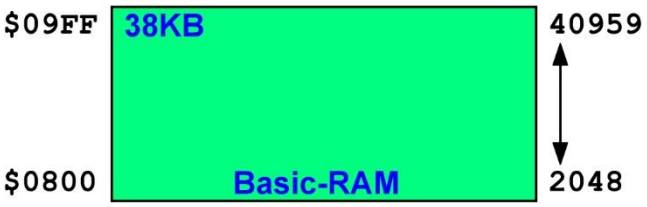


Abbildung 258 - Der Bereich für BASIC-Programme

An dieser Stelle ist auch ein wichtiger Hinweis zu beachten. Wenn man ein BASIC-Programm eintippt, dann ist es - nicht funktionell - wichtig, ob Leerzeichen eingegeben werden. Es macht also im Speicher einen Unterschied, ob

```
10 A=17
```

oder

```
10 A = 17
```

eingegeben wurde. Das sollte beachtet werden, denn dann können sich bei mir in den Beispielen genannte Speicheradressen etwas verschieben.

Sehen wir uns das folgende kleine Programm an.

Abbildung 259 - Ein einfaches BASIC-Programm

Die Frage ist nun, wie dieses Programm im Speicher ab der Adresse *$0800* (*2048*) abgelegt ist. Sind das genau die Zeichen, die hier als Listing zu sehen sind oder vielleicht doch in anderer Form? Man kann sich das natürlich genauer ansehen und wir nutzen dazu wieder SMON. Durch die Eingabe von ***M*** für Memory und den Speicheranfang bzw. -ende lassen wir uns den Inhalt anzeigen.

Die Eingabe von

M0800 081D

bringt folgendes zur Anzeige.

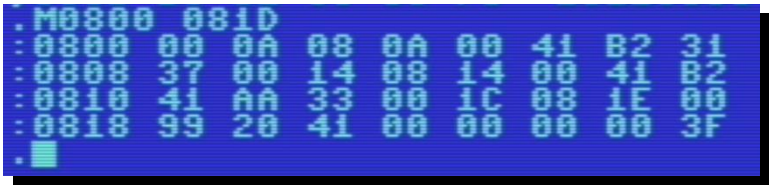


Abbildung 260 - Den Speicherinhalte im BASIC-RAM

Nun sind diese Werte ja in keiner Weise aussagekräftig und repräsentieren natürlich auch kein Assemblerprogramm. Ein Disassemblieren würde nur Quatsch hervorbringen. Hier ist eine andere Herangehensweise erforderlich. Beim C64 werden bei BASIC-Programmen die einzelnen Anweisungen, Zuweisungen oder andere Befehle (Schlüsselwörter) in den einzelnen Zeilen in Form von sogenannten *Tokens* (Schlüsselzeichen) verwaltet. Es wird dann zum Beispiel für eine *PRINT*-Anweisung nicht der ganze Befehl als Klartext im Speicher zu finden sein, sondern aus Gründen der Platzersparnis der Code *$99* hinterlegt. Werden im Direktmodus eingegebene BASIC-Befehle unmittelbar durch die Bestätigung über die *RETURN*-Taste bestätigt, werden diese vom BASIC-Interpreter, der auch *Tokenizer* genannt wird, über den Vorgang der Tokenisierung in ein Token oder mehrere Tokens umgewandelt.

Um also die gezeigten Werte des betreffenden Speicherinhaltes verstehen zu können, muss man sich verschiedener Tabellen bedienen. Zum einen ist da die Token-Tabelle, die zum Beispiel unter der folgenden Internetadresse zu finden ist.

	**Die Token-Tabelle**
https://www.c64-wiki.de/wiki/Token	

Natürlich gibt es nicht nur BASIC-Befehle, sondern auch Variablen und Werte vorhanden, für die es - diese Feststellung hätte ich mir auch schenken können - kein Token gibt. Derartige Informationen werden im Speicher über den sogenannten *PETSCII*-Code verwaltet.

Informationen zu diesem Code sind unter der folgenden Internetadresse zu finden ist.

Die PETSCII-Tabelle

- https://www.c64-wiki.com/wiki/PETSCII Codes in Listings
- https://c64os.com/post/c64petsciicodes

Wir wissen nun, dass das BASIC-RAM standardmäßig ab Adresse *$0800* beginnt. Diese Speicheradresse muss immer mit dem Wert *$00* belegt sein, denn ansonsten kommt es beim Start über *RUN* zu einer Fehlermeldung. Das eigentliche BASIC-Programm beginnt demnach eine Speicherstelle weiter, an Adresse *$0801*. Diese Adresse ist in einem sogenannten Zeiger mit dem Namen *TXTTAB* hinterlegt und ist für den BASIC-Programmstart zuständig. Es handelt sich um einen Zero-Page-Zeiger mit den Adressen *$2B/$2C* (*43/44*). Die Standardwerte sind demnach *$01/$08*. Doch sehen wir nach.

Abbildung 261 - Die Startadresse für BASIC-Programme ab $0801 (2049)

Sehen wir uns jetzt die Analyse der Werte im Speicher an. Doch bevor es sehr ins Detail geht, hier eine allgemeine Darstellung der Zusammenhänge der Programmzeilen.

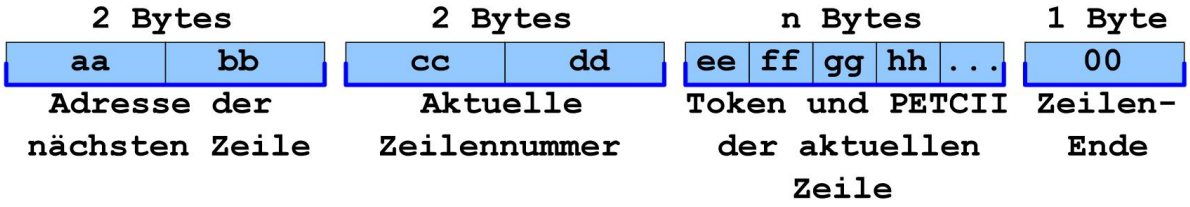


Abbildung 262 - Die einzelnen Bytes einer Befehlszeile

Jede der einzelnen Programmzeilen sind über einen Zeigermechanismus untereinander verknüpft, wobei immer eine Zeilennummer im Hintergrund auf die nachfolgende weist.

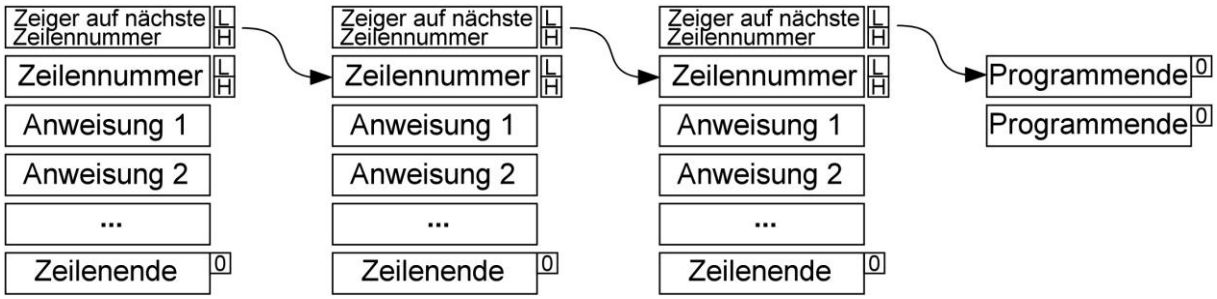


Abbildung 263 - Die allgemeine Verkettung der Programmzeilen

Zwischen der Zeilennummer einer Zeile befinden sich Anweisungen in Form von gespeicherten Tokens und Codes und eine abschließende Terminierung durch den Wert *0* als Zeilenende. Das Programmende erfolgt durch den zusätzlichen Anhang von zwei *0*-Bytes.

Diese drei Bytes (*$00, $00, $00*) im BASIC-Speicher markieren das Ende des BASIC-Programms. Jetzt wird es ernst und das komplette BASIC-Programm mit den drei Zeilen ist nachfolgend aufgeschlüsselt.

	Adresse HEX	Adresse Dez	Inhalt HEX	Inhalt Dez		Erläuterungen
	$0800	(2048)	$00	(0)		immer Null!
TXTTAB ⇨	$0801	(2049)	$0A	(10)	L	Pointer auf 2058
	$0802	(2050)	$08	(8)	H	(10 + 8 x 256 = 2058)
	$0803	(2051)	$0A	(10)	L	Zeilennummer 10
	$0804	(2052)	$00	(0)	H	(10 + 0 x 256 = 10)
	$0805	(2053)	$41	(65)		PETSCII-Code für A
	$0806	(2054)	$B2	(178)		Token für =
	$0807	(2055)	$31	(49)		PETSCII-Code für 1
	$0808	(2056)	$37	(55)		PETSCII-Code für 7
	$0809	(2057)	$00	(0)		Zeilenende
	$080A	(2058)	$14	(20)	L	Pointer auf 2068
	$080B	(2059)	$08	(08)	H	(20 + 8 x 256 = 2068)
	$080C	(2060)	$14	(20)	L	Zeilennummer 20
	$080D	(2061)	$00	(0)	H	(20 + 0 x 256 = 20)
	$080E	(2062)	$41	(65)		PETSCII-Code für A
	$080F	(2063)	$B2	(178)		Token für =
	$0810	(2064)	$41	(65)		PETSCII-Code für A
	$0811	(2065)	$AA	(170)		Token für +
	$0812	(2066)	$33	(51)		PETSCII-Code für 3
	$0813	(2067)	$00	(0)		Zeilenende
	$0814	(2068)	$1C	(28)	L	Pointer auf 2076
	$0815	(2069)	$08	(08)	H	(28 + 8 x 256 = 2076)
	$0816	(2070)	$1E	(30)	L	Zeilennummer 30
	$0817	(2071)	$00	(0)	H	(30 + 0 x 256 = 30)
	$0818	(2072)	$99	(153)		Token für PRINT
	$0819	(2073)	$20	(32)		PETSCII-Code für SPACE
	$081A	(2074)	$41	(65)		PETSCII-Code für A
	$081B	(2075)	$00	(0)		Zeilenende
	$081C	(2076)	$00	(0)	L	Programmende
	$081D	(2077)	$00	(0)	H	

Abbildung 264 - Das BASIC-Programm im Speicher

Ich möchte hier zum Abschluss des Themas eine Tabelle mit den Befehlen und entsprechenden Tokens zeigen.

Befehl	Token	Adresse	Befehl	Token	Adresse
END	$80 / 128	$A831	SPC(	$A6 / 166	$AAF9
FOR	$81 / 129	$A742	THEN	$A7 / 167	$A932
NEXT	$82 / 130	$AD1D	NOT	$A8 / 168	$AED4
DATA	$83 / 131	$A8F8	STEP	$A9 / 169	$A799
INPUT#	$84 / 132	$ABA5	+	$AA / 170	$B86A
INPUT	$85 / 133	$ABBF	-	$AB / 171	$B853
DIM	$86 / 134	$8081	*	$AC / 172	$BA2B
READ	$87 / 135	$AC06	/	$AD / 173	$BB12
LET	$88 / 136	$A9A5	^	$AE / 174	$BF7B
GOTO	$89 / 137	$A8A0	AND	$AF / 175	$AFE9
RUN	$8A / 138	$A871	OR	$B0 / 176	$AFE6
IF	$8B / 139	$A928	>	$B1 / 177	$BFB4
RESTORE	$8C / 140	$A81D	=	$B2 / 178	$AED4
GOSUB	$8D / 141	$A883	<	$B3 / 179	$B016
RETURN	$8E / 142	$A8D2	SGN	$B4 / 180	$BC39
REM	$8F / 143	$A93B	INT	$B5 / 181	$BCCC
STOP	$90 / 144	$A82F	ABS	$B6 / 182	$BC58
ON	$91 / 145	$A94B	USR	$B7 / 183	$0310
WAIT	$92 / 146	$B82D	FRE	$B8 / 184	$B37D
LOAD	$93 / 147	$E168	POS	$B9 / 185	$B39E
SAVE	$94 / 148	$E156	SQR	$BA / 186	$BF71
VERIFY	$95 / 149	$E165	RND	$BB / 187	$E097
DEF	$96 / 150	$B3B3	LOG	$BC / 188	$B9EA
POKE	$97 / 151	$B824	EXP	$BD / 189	$BFED
PRINT#	$98 / 152	$AA80	COS	$BE / 190	$E264
PRINT	$99 / 153	$AAA0	SIN	$BF / 191	$E26B
CONT	$9A / 154	$A857	TAN	$C0 / 192	$E2B4
LIST	$9B / 155	$A69C	ATN	$C1 / 193	$E30E
CLR	$9C / 156	$A65E	PEEK	$C2 / 194	$B80E
CMD	$9D / 157	$AA86	LEN	$C3 / 195	$B77C
SYS	$9E / 158	$E12A	STR$	$C4 / 196	$B465
OPEN	$9F / 159	$E1BE	VAL	$C5 / 197	$B7AD
CLOSE	$A0 / 160	$E1C7	ASC	$C6 / 198	$B78B
GET/GET#	$A1 / 161	$AB7B	CHR$	$C7 / 199	$B6EC
NEW	$A2 / 162	$A642	LEFT$	$C8 / 200	$B700
TAB(	$A3 / 163	$AAE8	RIGHT$	$C9 / 201	$B72C
TO	$A4 / 164	$A76D	MID$	$CA / 202	$B737
FN	$A5 / 165	$B3F4	GO	$CB / 203	$A812

Tabelle 44 - Token

Jedes Token hat eine sogenannte Ausführungsadresse (EXEC) im ROM, wo der entsprechende Code für den jeweiligen Token bzw. BASIC-Befehl ausgeführt wird.

Die Speicherung der Daten

- Die Fließkomma-Variablen
- Die Integer-Variablen
- Die String-Variablen
- Die Array-Variablen
- Eine Funktion

Die Speicherung der Daten

Wir haben gesehen, dass ein Computerprogramm nicht nur aus dem Quellcode besteht, sondern auch Daten zur späteren Verarbeitung speichert. Es sind die vier Variablen- bzw. Datentypen verfügbar

Datentyp	Kennzeichnung	Beispiel	Inhalte
Fließkomma	keine	A	numerisch
Integer	%	A%	numerisch - Ganzzahl
String	$	A$	alphanumerisch
Array	DIM	DIM A%(2)	für alle Datentypen mit Angabe des Index

Tabelle 45 - Die Datentypen im Überblick

Unser sehr einfaches Programm, das ich nun genauer beleuchten möchte, besteht aus der Befehlszeile *INPUT B*. Nach dessen Start wird nach dem Wert für *B* gefragt, damit dieser dann im Anschluss ebenfalls im Speicher abgelegt werden kann. Programm und Daten dürfen sich aber nicht in die Quere kommen und somit werden die Daten in einem besonderen Bereich des Arbeitsspeichers abgelegt, was bei den einfachen Variablen unmittelbar im Anschluss des eigentlichen Programms erfolgt. Es ist sicherlich verständlich, dass diese Bereiche nicht statisch sind, denn Programme können, je nach Umfang, kleiner oder größer sein und das ist natürlich auch auf die Variablen zu übertragen. Dieses Verhalten wird durch die senkrechten Pfeile, die sich rechts neben den Blockbezeichnungen befinden, angezeigt. Sehen wir uns den entsprechenden Teil des Arbeitsspeichers noch einmal genauer an.

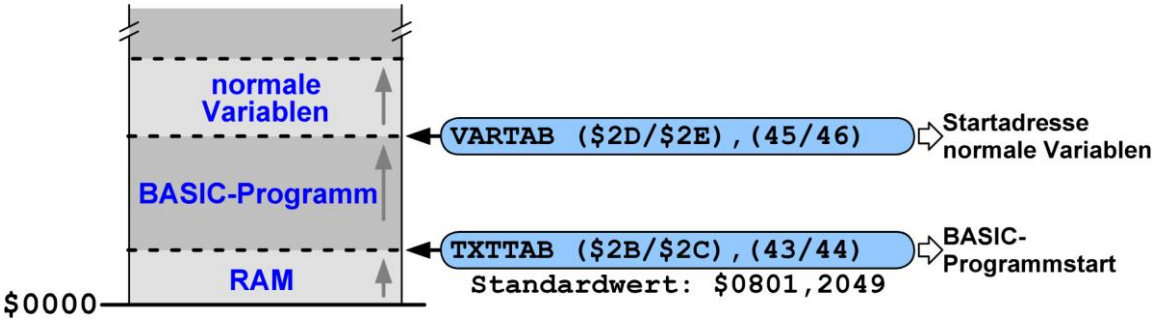


Abbildung 265 - Der VARTAB-Pointer am oberen Rand

Für unser einfaches Programm mit nur einer einzigen Zeile schaut das dann wie folgt aus.

```
10 INPUT B
READY.
?PEEK(45)+256*PEEK(46)
 2059

READY.
```

Abbildung 266 - Das einfache BASIC-Programm für die Variable B

Ok, ab Adresse *2059*, was dem hexadezimalen Wert *$080B* entspricht, fängt der Bereich für die normalen Variablen an. Sehen wir dort einmal nach, nachdem wir das Programm gestartet und den Wert *3.14* eingegeben haben. Jetzt ist übrigens klar, dass es sich um einen Fließkommawert handelt.

Abbildung 267 - Das BASIC-Programm wurde gestartet

Nun ist es soweit, mit SMON einmal nachzusehen, wie sich dort die Werte gestalten. Wir starten SMON wieder über

SYS 49152

und geben dort den Memory-Befehl

M080B 081A

ein.

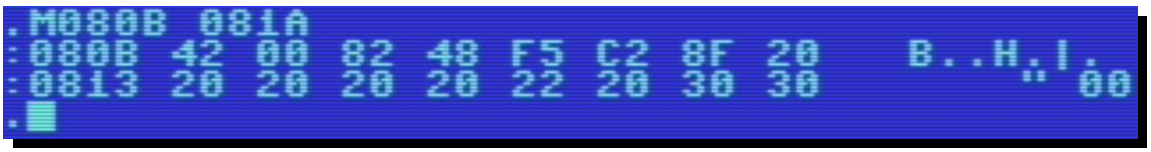


Abbildung 268 - Hier soll die Variable B zu finden sein

Den Namen der Variablen können wir schon in der ersten Adresse sehen, denn dort ist der Wert *$42* zu sehen und rechts außen befindet sich gleich die Interpretation des PETSCII. Nur der Wert *3.14* ist mit dieser Methode über den PETSCII nicht auszumachen. Kommen wir also zum ersten Unterthema dieses Kapitels, wenn es um die Speicherung von Fließkommawerten geht.

Die Fließkomma-Variablen

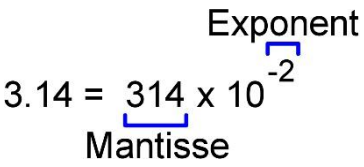
Bei Fließ- oder Gleitkommazahlen handelt es sich um Zahlen, die sehr groß oder sehr klein werden können und die sich mit Ganzzahlen nicht darstellen lassen, da sie gebrochene Werte enthalten. Hinsichtlich eines Variablennamen wird dieser ohne einen Zusatz angegeben. Also zum Beispiel

A = 23

B = 17.5

Es müssen also Werte gespeichert werden, die ein Komma enthalten wie zum Beispiel die Kreiszahl ***PI*** mit dem hier sehr verkürzten Wert ***3,14***. Es sei zu bemerken, dass in der Programmierung anstelle des Kommas ein Punkt verwendet wird, so dass der Wert ***3.14*** lautet. Wir werden jetzt sehen, wie sich dieser Wert im

Speicher zeigt. Die Darstellung von Fließkommazahlen ist nach der Norm *IEEE 754* standardisiert. Eine Fließpunktzahl ist aus der *Mantisse* und dem *Exponenten* zusammengesetzt.



Nachfolgend ist die Darstellung der einzelnen Bytes für eine Floatingpoint-Variable zu sehen.

Nachfolgend ist die Darstellung der einzelnen Bytes für eine Floatingpoint-Variable zu sehen.

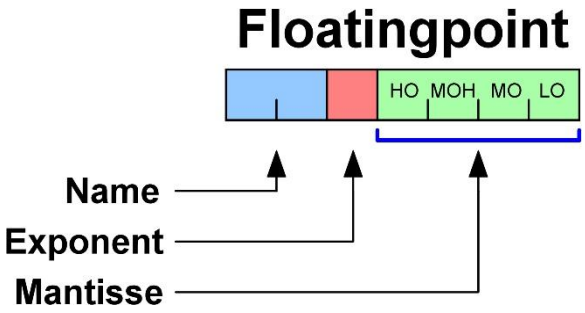


Abbildung 269 - Die 7 Bytes des Floatingpoint-Datentyps

Die weitere Betrachtung der beiden führenden Bytes, die für den Variablennamen stehen, werde ich nicht weiter fortführen. Kommen wir zu den restlichen *5* Bytes, die den eigentlichen Inhalt repräsentieren. Bei Gleitkommazahlen besteht der Inhalt aus einem *Ein-Byte Exponenten*, gefolgt von einer vier Byte langen Mantisse. Der Exponent besitzt ein besonderes Format und liegt in der sogenannten *Excess-$80*-Form vor. Man redet hier von einem Offset- oder BIAS-Wert, wobei der Wert *$80* (*128*) der eigentliche Ausgangspunkt ist und in diesem Format für einen Nullexponenten steht. Die weiteren Werte wie zum Beispiel *$81*, *$82* und *$83* repräsentieren die positiven Exponenten von *1*, *2* und *3*. Werte unterhalb von *$80*, also *$7F*, *$7E* oder *$7D* werden als negative Exponenten *-1*, *-2* und *-3* erkannt.

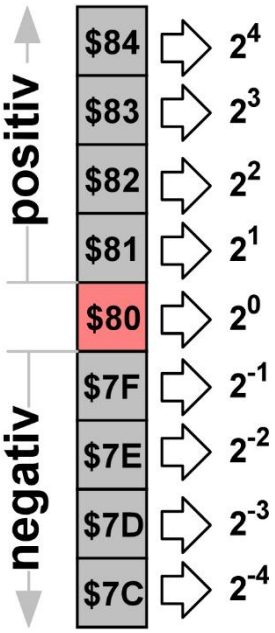


Abbildung 270 - Der Exponent in der Excess-$80-Form

Kommen wir zur Mantisse, die ja als Faktor vor der Basis mit dem Exponenten steht. Die Mantisse ist normalisiert und tritt in abnehmender Reihenfolge in Erscheinung. Die vier Bytes *HO*, *MOH*, *MO*, *LO* werden in der gezeigten Reihenfolge von links nach rechts aufgeführt. Der Binärpunkt befindet sich unmittelbar links von HO. Aufgrund der Normalisierung besitzt das MSB (Bit 7) des HO-Bytes immer den Wert *1* und trägt demnach keine verwertbare Information. Dennoch spielt es für uns eine entscheidende Rolle, denn es wird bei der Speicherung des Variablenwertes als Vorzeichenbit der Mantisse genutzt und ist demnach nicht zu unterschlagen! Folgendes gilt.

- Bit 7 = 0: Es handelt sich um einen positiven Wert
- Bit 7 = 1: Es handelt sich um einen negativen Wert

Durch dieses Verfahren wird jedes einzelne Bit genutzt und wirkt dem Vorhandensein überflüssiger Bits entgegen, denn Speicherplatz ist rar. Dieses Vorgehen wird als *Packing* (deutsch: packen) bezeichnet. Eine Variable, die als Floatingpoint definiert wurde und der *5* Bytes zur Verfügung stehen, kann Werte im Bereich von *+/- 2.9387355E-39* bis *+/- 1.70141183E+38* speichern. Nun genug der Vorrede, denn es wird Zeit, mit einem konkreten Beispiel zu starten, denn ich erwähnte ja schon, dass wir den Wert *3.14* speichern wollen, um dann zu sehen, wie sich der Speicherinhalt gestaltet und ob wir aufgrund dieser Werte und deren Analyse wieder auf den eigegebenen Wert kommen. Das Programm ist schon gekannt und wir können es unverändert in dieser einen Zeile übernehmen, denn der Datentyp ist schon korrekt gewählt, denn ohne ein Prozentzeichen am Ende des Namens, handelt es sich um eine Floatingpoint-Variable.

Nun geht es daran, nachzusehen, was der Speicher so hergibt. Die Ausgabe haben wir schon gesehen, doch ich werde jetzt die einzelnen Bytes farbig hervorheben, damit es etwas klarer wird.

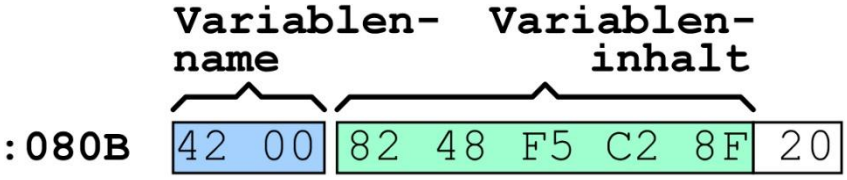


Abbildung 271 - Die Variable B im Speicher mit dem Wert +3.14

In Blau ist der Variablenname zu sehen. Der PETSCII-Wert *$42* steht für den Buchstaben *B* mit dem Dezimalwert *66*. Die nachfolgende Speicherstelle ist *0*, was bedeutet, dass der Variablenname nur aus einem einzigen Buchstaben besteht.

Doch hier gibt es etwas zu beachten. Sehen wir uns dazu das Byte *1* und das Byte *2* an, denn es ist nicht nur der Variablenname kodiert, sondern auch der Datentyp, der auf eine Fließkommavariable (Floatingpoint) hinweist. Das Bit *7* der beiden Bytes sind beide auf *0* gesetzt sind und signalisieren, dass es sich um eine Fließkommazahl handelt.

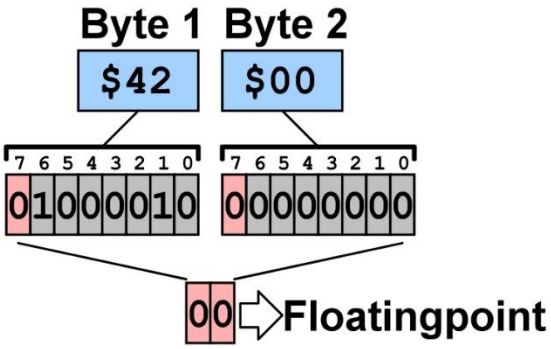


Abbildung 272 - Die beiden Bytes (Byte 1 und Byte 2) im Detail

Doch zurück zu den restlichen Bytes. Die nachfolgenden Werte *$82*, *$48*, *$F5*, *$C2* und *$F8* werde ich jetzt zur besseren Übersicht hinsichtlich der einzelnen Bits auseinanderziehen und größer darstellen.

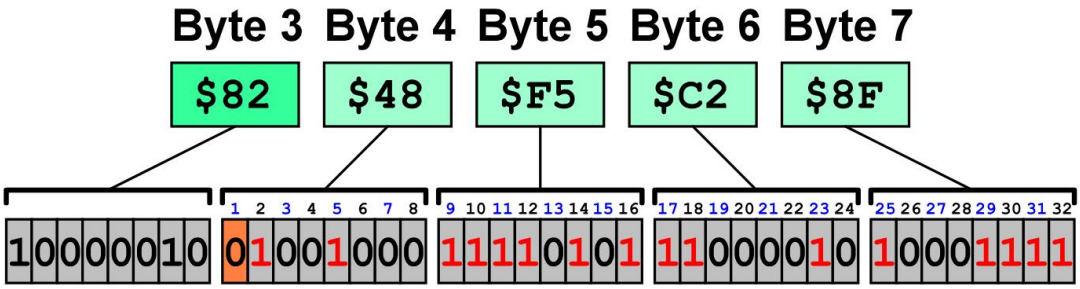


Abbildung 273 - Die Bytes und Bits des Variablenwertes +3.14

Wie muss man nun vorgehen? Um die Dezimalzahl aus dem zu sehenden Binärformat zu berechnen, multipliziert man den Wert jedes Bits der Mantisse mit dem entsprechenden Binärwert und addiert am

Ende den *0.5* hinzu, was eine dezimale Entsprechung des binären Wertes *0.1* ist. Abschließend muss das Ergebnis mit dem Wert des Exponenten minus *128* multipliziert werden. Es ist weiterhin zu bemerken, dass ich die Nummerierung der einzelnen Bits der Mantisse von links nach rechts vollzogen habe. Man muss das nicht machen, doch so erleichtert es uns die Übersicht beim Rechnen. Damit es noch übersichtlicher wird, habe ich die Eins-Werte in Rot hervorgehoben, denn nur diese Stellen müssen bei der Aufsummierung berücksichtigt werden. Los geht's!

Der Exponent - Byte 3

Da der Wert des Exponenten hier *$82* ist, muss aufgrund des genannten Formates der Wert *$80* subtrahiert werden, so dass sich ein Endergebnis für den Exponenten für die spätere Rechnung von *2* ergibt.

Die Summierung von Byte 4

$0.5^2 + 0.5^5 = 0.28125$

Die Summierung von Byte 5

$0.5^9 + 0.5^{10} + 0.5^{11} + 0.5^{12} + 0.5^{14} + 0.5^{16} = 0.003738403$

Die Summierung von Byte 6

$0.5^{17} + 0.5^{18} + 0.5^{23} = 0.000011563$

Die Summierung von Byte 7

$0.5^{25} + 0.5^{29} + 0.5^{30} + 0.5^{31} + 0.5^{32} = 0.000000033$

Ich sollte an dieser Stelle aufgrund von Ungenauigkeiten raten, die komplette Rechnung der Aufsummierungen in einem Rutsch durchzuführen.

Die Endrechnung

Ich summiere jetzt die Einzelsummen auf, addiere *0.5* hinzu und multipliziere das Ergebnis wiederum mit 2^2. Als Ergebnis erhalte ich *3.139999997*, was schon nahe an *3.14* kommt. Mit dem Überspringen der Teilsummenbildung erhalten wir als Ergebnis wirklich *3.14*.

Was passiert eigentlich, wenn wir statt *3.14* den negativen Wert -3.14 eingeben? Was ändert sich an den Bytes und Bits? Sehen wir nach.

Abbildung 274 - Der Variablenwert wurde auf -3.14 geändert

Zur besseren Übersicht hebe ich die Änderung wieder hervor.

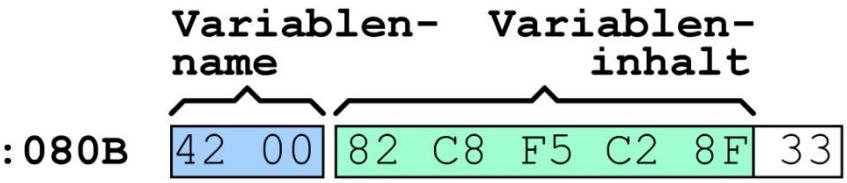


Abbildung 275 - Die Variable B im Speicher mit dem Wert -3.14

In Blau ist natürlich wieder der Variablenname zu sehen. Der PETSCII-Wert *$42* steht für den Buchstaben *B* mit dem Dezimalwert *66*. Die nachfolgende Speicherstelle ist *0*, was bedeutet, dass der Variablenname nur aus einem einzigen Buchstaben besteht. Daran hat sich natürlich zum vorherigen Beispiel nichts geändert! Doch der Wert *$48* von Byte *4* ist auf *$C8* gesprungen. Hier die Details dazu.

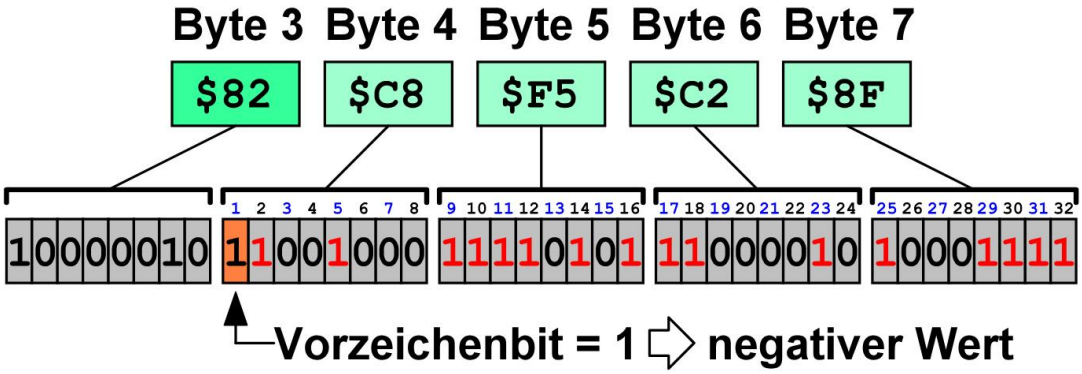


Abbildung 276 - Die Bytes und Bits des Variablenwertes -3.14

Es ist zu sehen, dass sich lediglich das Vorzeichenbit in Byte *4* von *0* auf *1* geändert hat und darüber einen negativen Wert kennzeichnet. Eine Umrechnung des Wertes wie bei den Ganzzahlen über die Bildung des Zweierkomplementes ist also nicht erforderlich.

Die Integer-Variablen

Kommen wir jetzt zu den Integer-Variablen, die ja bekanntlich nur Ganzzahlen speichern können. Hinsichtlich eines Variablennamen wird dieser mit eine Prozentzeichen (**%**) angegeben. Also zum Beispiel

```
A% = 456

B% = 17
```

Der Wertebereich einer derartigen Variablen liegt zwischen *-32767* und *+32767*. Das vorherige Programm müssen wir lediglich dahingehend ändern, dass dem Variablennamen ein Prozentzeichen angehängt wird.

Ich habe absichtlich einmal einen zu hohen Wert eingegeben, der nicht in den vorgegebenen Wertebereich passt, was mit einem

?ILLEGAL QUANTITY ERROR

quittiert wird.

```
10 INPUT B%
RUN
? 32768

?ILLEGAL QUANTITY  ERROR IN 10
READY.
```

Abbildung 277 - Das Programm zur Speicherung einer Integer-Variablen

Ok, ich starte das Programm erneut und geben den Wert *27389* ein. Dann wollen wir sehen, wie sich das im Speicher äußert. Zuerst sehen wir uns wieder den Speicherinhalt an und dann die Erklärungen dazu.

```
10 INPUT B%
READY.
RUN
? 27389

READY.
```

Abbildung 278 - Die Integer-Variable mit einem legalen Wert

Sehen wir uns das Programm im Speicher an.

	Adresse HEX	Adresse Dez	Inhalt HEX	Inhalt Dez		Erläuterungen
	$0800	(2048)	$00	(0)		immer Null!
TXTTAB ⇨	$0801	(2049)	$0A	(10)	L	Pointer auf 2058 (10 + 8 x 256 = 2058)
	$0802	(2050)	$08	(8)	H	
	$0803	(2051)	$0A	(10)	L	Zeilennummer 10 (10 + 0 x 256 = 10)
	$0804	(2052)	$00	(0)	H	
	$0805	(2053)	$85	(133)		Token für INPUT
	$0806	(2054)	$20	(32)		PETSCII-Code für SPACE
	$0807	(2055)	$42	(66)		PETSCII-Code für B
	$0808	(2056)	$25	(37)		PETSCII-Code für %
	$0809	(2057)	$00	(0)		Zeilenende
	$080A	(2058)	$00	(0)	L	Programmende
	$080B	(2059)	$00	(0)	H	

Abbildung 279 - Das Programm im Speicher

Können wir eigentlich sicher sein, dass die Startadresse der Daten die gleiche ist, wie beim vorherigen Beispiel? Dort lautete sie *$080B* also in dezimaler Schreibweise *2059*. Sehen wir doch hier einmal nach.

Wie lautet der VARTAB-Pointer?

```
?PEEK(45)+256*PEEK(46)
 2060

READY.
```

Abbildung 280 - Der VARTAB-Pointer

Ok, dieser hat sich geändert und liegt bei dezimal *2060*, was hexadezimal *$080C* bedeutet. Das hängt damit zusammen, dass sich die Daten in der Regel unmittelbar im Anschluss des Programmendes ansiedeln. Die letzte Speicherstelle im Programm war die Adresse *$080B* und im Anschluss wäre dann *$080C* frei. Dann starten wir SMON und sehen uns den entsprechenden Speicherbereich an und geben den Befehl

M080C

ein.

```
.M080C
:080C C2 80 6A FD 00 00 00 20   |.......
.
```

Abbildung 281 - Die Anzeige der Daten für die Integer-Variable

Was hier vielleicht sofort auffällt, ist die Tatsache, dass der Variablenname rechts neben den acht Bytes nicht zu erkennen ist. Dort stand zuvor an erster Stelle doch das ***B***. Das hat eine besondere Bewandtnis, auf die ich sofort zu sprechen komme. Sehen wir uns zuvor die Darstellung der einzelnen Bytes für eine Integer-Variable an.

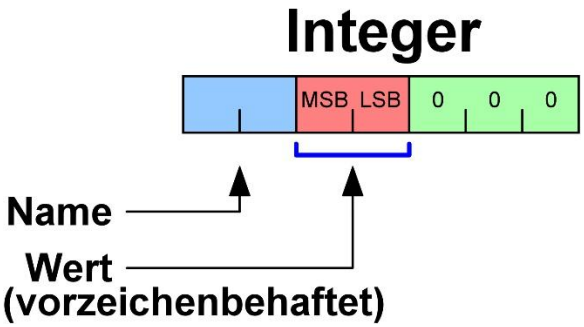


Abbildung 282 - Die 4 Bytes des Integer-Datentyps

Die weitere Betrachtung der beiden führenden Bytes, die für den Variablennamen stehen, werde ich diesmal besonders unter die Lupe nehmen. Die beiden nachfolgenden Bytes stehen für das *MSB* und für das *LSB*. Die restlichen drei Bytes werden hier nicht genutzt. Kommen wir auf den Namen der Variablen zu sprechen, der ja immer noch *B* lautet, auch, wenn es sich diesmal um eine Integer-Variable handelt. Der Wert ist jedoch nicht *$42*, sondern *$C1* und der zweite Wert ist nicht *$00*, sondern jetzt *$80*.

Nun geht es daran, nachzusehen, was der Speicher wieder so hergibt. Die Ausgabe haben wir schon gesehen, doch ich werde jetzt die einzelnen Bytes farbig hervorheben, damit es etwas klarer wird.

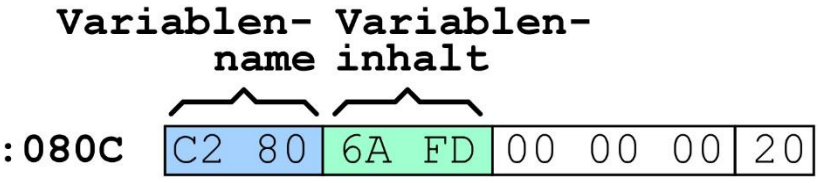


Abbildung 283 - Die Variable B im Speicher mit dem Wert 27389

In Blau ist der Variablenname zu sehen. Die PETSCII-Werte helfen an dieser Stelle nicht weiter, weil innerhalb des Variablennamen eine Kodierung stattfindet, die auf den Datentyp hinweist. Bei dem Fließkommawert waren die entsprechenden Bits gleich Null und deswegen konnte der PETSCII-Wert unmittelbar abgelesen werden. Hier schaut das anders aus, denn die entsprechenden Bits sind ungleich Null. Das Bit *7* der beiden Bytes sind beide auf *1* gesetzt sind und signalisieren, dass es sich um eine Integerzahl handelt.

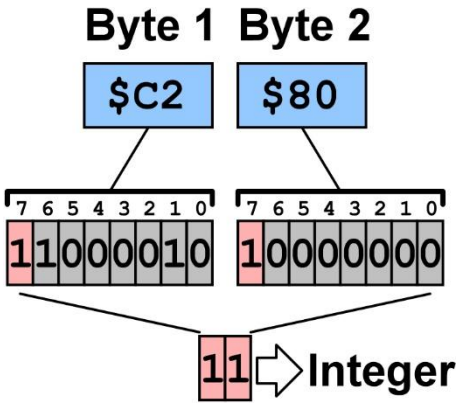


Abbildung 284 - Die beiden Bytes (Byte 1 und Byte 2) im Detail

Warum wird aber immer noch der Variablenname *B* erkannt, denn es ist doch nicht *$42* in Byte *1* zu sehen. Wir erinnern uns, dass der ASCII-Code bzw. PETSCII-Code nur 7-Bit breit ist. Auf der nachfolgenden Abbildung ist des Rätsels Lösung zu sehen. Wird das MSB nicht berücksichtigt, kommt man auf die ursprünglichen Werte.

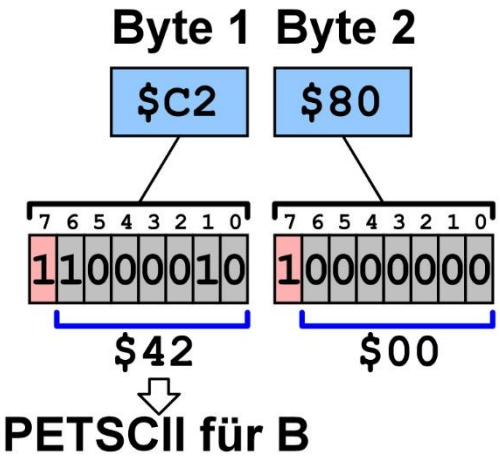


Abbildung 285 - Die Dekodierung des Variablennamen 'B'

Ok, dann sehen wir uns zum Schluss noch die letzten beiden Bytes *3* und *4* an.

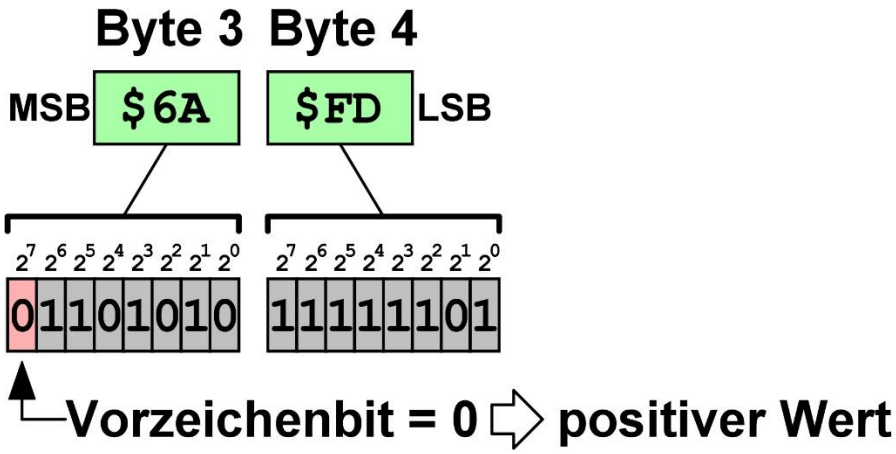


Abbildung 286 - Die Ermittlung des Integer-Wertes

Das Ergebnis lautet dann korrekterweise.

$$Integerwert = LSB + 256 \cdot MSB = 253 + 256 \cdot 106 = 27389$$

Wir sollten jetzt einen Blick auf negative Integerwerte werfen, denn diese müssen ja auch irgendwie gekennzeichnet sein, damit sie als solche erkannt werden. Es muss also ebenfalls ein Vorzeichen-Bit vorhanden sein, um negative Werte von positiven zu unterscheiden. Und dem ist natürlich auch so. Wir kennen das schon. Auch hier gilt die Kennzeichnung, dass ein Vorzeichen-Bit mit einer *0* einen positiven und mit einer *1* einen negativen Wert markiert.

Ich habe also unser Programm noch einmal gestartet und dann den negativen Wert von *-27389* eingegeben. Mal sehen, was denn dabei herausgekommen ist.

```
RUN
? -27389

READY.
```

Abbildung 287 - Die Eingabe eines negativen Wertes

Lassen wir wieder SMON sprechen und sehen, was sich bei den Bytes so tut und sehen uns den entsprechenden Speicherbereich an und geben den Befehl

M080C

ein.

```
.M080C
:080C C2 80 95 03 00 00 00 20    |.......
.
```

Abbildung 288 - Die Anzeige der Daten für die Integer-Variable

Das müssen wir uns wieder im Detail ansehen.

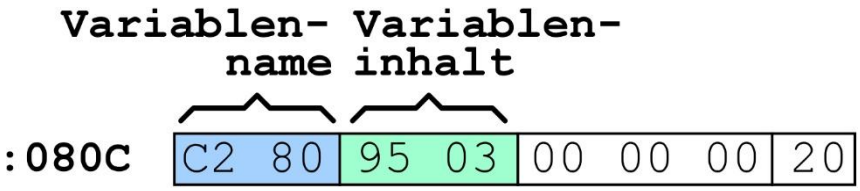


Abbildung 289 - Die Variable B im Speicher mit dem Wert -27389

Die beiden Werte *$95* und *$03* müssen also den Wert *-27389* repräsentieren. Sehen wir nach, denn das Spiel mit dem Einer- und Zweierkomplement sollte mittlerweile klar sein.

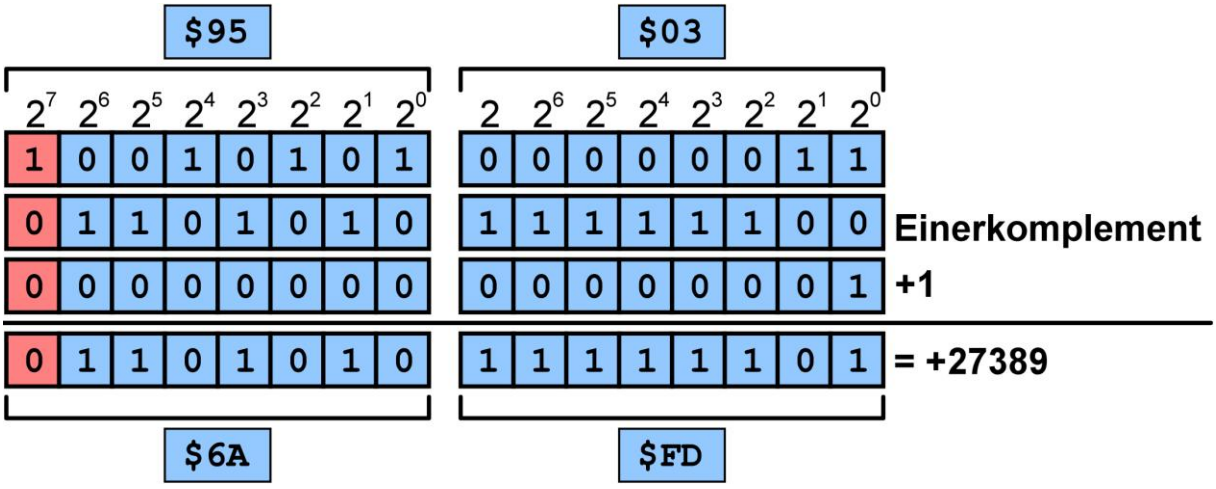


Abbildung 290 - Die Bildung des Zweierkomplements

Das schaut doch richtig aus!

Die String-Variablen

Bei String-Variablen handelt es sich um Zeichenketten, also einer Ansammlung von ASCII kodierten Zeichen, die in doppelten Hochkomma eingeschlossen sind. Bei einer derartigen Variablen muss ein Dollar-Zeichen (**$**) dem Variablennamen angefügt werden. Also zum Beispiel *A$*, *B$* oder *HO$*. Sehen wir uns dazu das folgende Programm und die Eingabe einer Textzeile an.

```
10 INPUT A$
READY.
RUN
? HIER SPRICHT EIN C64

READY.
```

Abbildung 291 - Das Programm zur Speicherung einer Zeichenkette

Es wurde also der Text *„HIER SPRICHT EIN C64"* eingegeben, der in der Variablen *A$* gespeichert wurde. Da diese Zeichen - wie schon erwähnt - ASCII kodiert sind, muss demnach die folgende Bytefolge irgendwo im Speicher zu finden sein. Da es sich diesmal um eine Zeichenkette handelt, müssen wir einen bestimmten Pointer überprüfen, der sich *FRETOP* nennt.

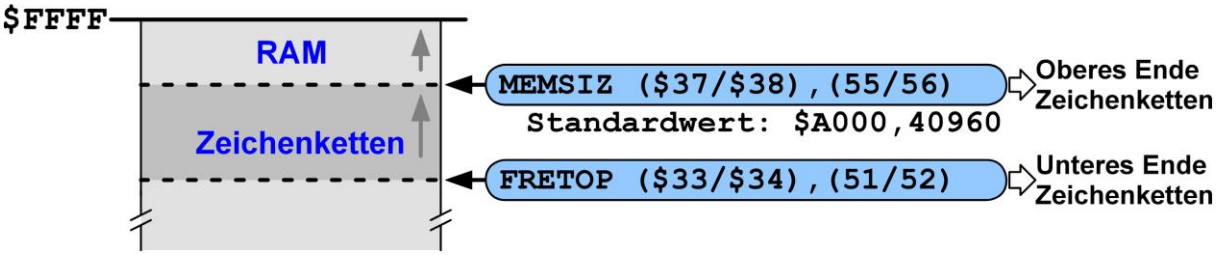


Abbildung 292 - Der FRETOP-Pointer

Sehen wir in BASIC nach, auf welche Adresse dieser Pointer zeigt.

```
?PEEK(51)+256*PEEK(52)
 40940

READY.
```

Abbildung 293 - Die FRETOP-Adresse

Es handelt sich um den dezimalen Wert *40940*, der in hexadezimaler Schreibweise *$9FEC* lautet und das geben wir dann mal in SMON zur Speicheranzeige ein. Der Befehl lautet

M9FEC 9FFF

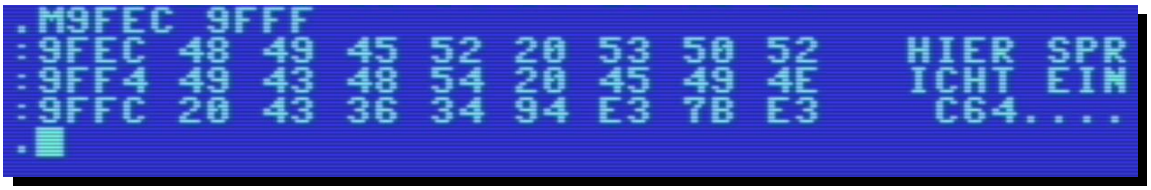


Abbildung 294 - Den FRETOP-Speicherbereich anzeigen

Hier sehen wir auch schon die Interpretation des PETSCII-Codes, der mit der vorherigen Eingabe übereinstimmt. Aber wir sollten einen Blick in das Programm werfen, damit wir erkennen, warum diese Adresse *$9FEC* (*40940*) für den Anfang der Zeichenkette passend ist. Also rufen wir zuerst wieder den Bereich auf, der für ein BASIC-Programm relevant ist. Ich habe dazu die wichtigen Bereiche etwas farblich hervorgehoben. Der hellblaue Bereich steht für das eigentliche Programm, das wir uns gleich ansehen werden und der hellgelbe Bereich für die Information zur Variablen A$, in der die Zeichenkette gespeichert wird.

Abbildung 295 - Der komplette Speicherbereich für das Programm + Daten

Die Interpretation des Programms schaut wie folgt aus.

	Adresse HEX	Adresse Dez	Inhalt HEX	Inhalt Dez		Erläuterungen
	$0800	(2048)	$00	(0)		immer Null!
TXTTAB ⇨	$0801	(2049)	$0A	(10)	L	Pointer auf 2058 (10 + 8 x 256 = 2058)
	$0802	(2050)	$08	(8)	H	
	$0803	(2051)	$0A	(10)	L	Zeilennummer 10 (10 + 0 x 256 = 10)
	$0804	(2052)	$00	(0)	H	
	$0805	(2053)	$85	(133)		Token für INPUT
	$0806	(2054)	$20	(32)		PETSCII-Code für SPACE
	$0807	(2055)	$41	(65)		PETSCII-Code für A
	$0808	(2056)	$24	(36)		PETSCII-Code für $
	$0809	(2057)	$00	(0)		Zeilenende
	$080A	(2058)	$00	(0)	L	Programmende
	$080B	(2059)	$00	(0)	H	

Abbildung 296 - Das BASIC-Programm im Speicher

Die hier zu sehenden Informationen sollten eigentlich bekannt sein. Wichtig sind die Daten, die sich direkt im Anschluss an das Programm befinden und die ich in hellgelb markiert habe. Wichtig zum Verständnis der abgespeicherten Zeichenkette sind jetzt die Werte im Speicher. Nachfolgend ist die Darstellung der einzelnen Bytes für eine String-Variable zu sehen.

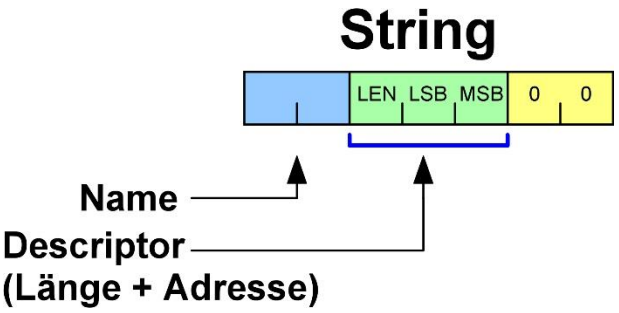


Abbildung 297 - Die 5 Bytes des String-Datentyps

Es versteht sich von selbst, dass Zeichenketten nicht in sieben Bytes untergebracht werden können. Um eine Flexibilität bei der Zuweisung von Strings zu ermöglichen - ohne ihre Länge angeben zu müssen, wie es in einigen Programmiersprachen erforderlich ist -, werden sie dynamisch im RAM gespeichert.

Die weitere Betrachtung der beiden führenden Bytes, die für den Variablennamen stehen, müssen wir wieder unter die Lupe nehmen. Die drei nachfolgenden Bytes stehen für die Länge der Zeichenkette (*LEN*) und ein Zeiger auf die eigentliche Zeichenkette über das *LSB* und das *MSB*. Die restlichen zwei Bytes werden hier nicht genutzt. Kommen wir auf den Namen der Variablen zu sprechen, der ja jetzt *A$* lautet.

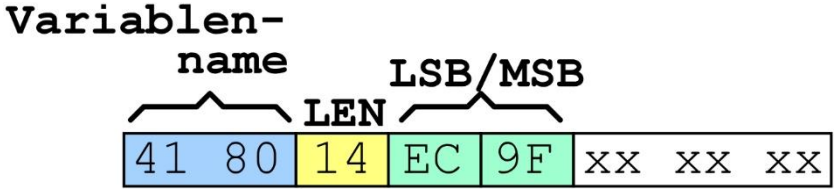


Abbildung 298 - Die Variable A$ im Speicher

Sehen wir uns wieder die einzelnen Bytes genauer an. Zuerst den Variablennamen *A$*. Über das Bit *7* der Bytes *1* und *2* ist der Datentyp String kodiert.

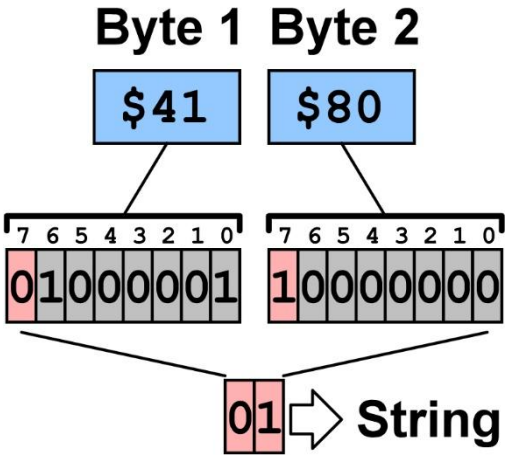


Abbildung 299 - Der String-Datentyp versteckt sich im Namen der Variablen

Ok, dann sehen wir uns zum Schluss noch die letzten drei Bytes *3*, *4* und *5* an.

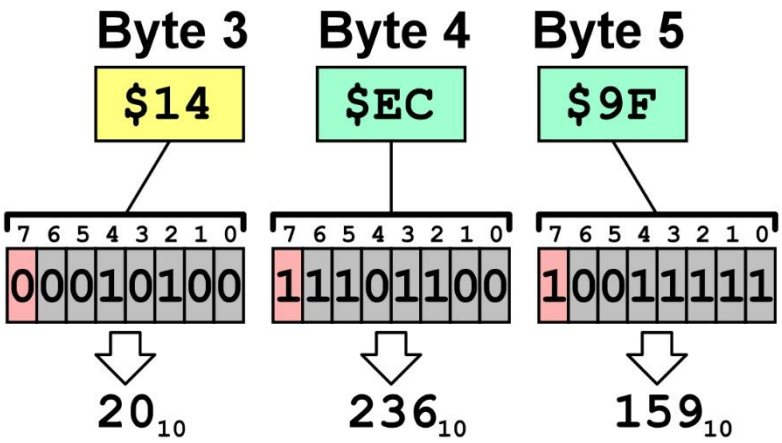


Abbildung 300 - Die Bytes und Bits für Länge und Pointer der Zeichenkette

Der Wert in Byte 3 gibt die Länge (LEN = Length) der Zeichenkette an und schlägt in unserem Fall mit *20* Zeichen zu Buche. Die beiden darauffolgenden Bytes stehen für das LSB und MSB als Zeiger auf die Zeichenkette im Speicher, die wir eben schon an Adresse *$9FEC* (*40940*) ausfindig gemacht haben. Stimmt das auch?

$$Pointer = LSB + 256 \cdot MSB = 236 + 256 \cdot 159 = 40940$$

Das passt also! Ab der Speicherstelle *40940*, in HEX *$9FEC* ist also der Anfang der Zeichenkette zu finden.

Die Array-Variablen

Ein Array ist eine besondere Variable, die mehrere Werte des gleichen Datentyps enthalten kann. Sind zum Beispiel 10 Integer-Variablen notwendig, die Messwerte aufnehmen sollen, dann ist es ratsam, eine Array-Variable zu definieren. Die einzelnen Elemente des Arrays sind einfach über einen Index anzusprechen und abzurufen beziehungsweise mit neuen Daten zu versehen. Es ist zu beachten, dass die Zählung des Index bei *0* beginnt. Über das folgende Programm möchte ich *3* Werte in einem Array vom Integer-Datentyp speichern.

```
10 DIM A%(2)
20 FOR I = 0 TO 2
30 INPUT A%(I)
40 NEXT I
READY.
```

Abbildung 301 - Das Programm zur Speicherung eines Arrays

Es ist zu sehen, dass in Zeile *20* eine Array-Variable mit runden Klammern versehen wurde, in denen sich der höchste Index befindet, der angesprochen werden darf. Für *3* Elemente ist also die Dimensionierung mit *A%(2)* zu versehen, da die Zählung über den Index bei *0* beginnt. Wer das nicht möchte, kann natürlich die *0* weglassen und dann über *A%(3)* von Index *1* bis *3* die Elemente zu adressieren. Natürlich bedeutet das ein wenig Speicherverschwendung. Sehen wir uns zunächst das Programm im

Speicher an, und diesmal wird es etwas umfangreicher, denn wir haben ja schon *4* Programmzeilen. Werfen wir zuerst einen Blick auf das Programm im Speicher, bevor es an die Analyse der Werte geht.

Adresse HEX	Adresse Dez	Inhalt HEX	Inhalt Dez		Erläuterungen
$0800	(2048)	$00	(0)		immer Null!
$0801	(2049)	$0D	(13)	L	Pointer auf 2061
$0802	(2050)	$08	(8)	H	*(13 + 8 x 256 = 2061)*
$0803	(2051)	$0A	(10)	L	Zeilennummer 10
$0804	(2052)	$00	(0)	H	*(10 + 0 x 256 = 10)*
$0805	(2053)	$86	(134)		Token für DIM
$0806	(2054)	$20	(32)		PETSCII für SPACE
$0807	(2055)	$41	(65)		PETSCII-Code für A
$0808	(2056)	$25	(37)		PETSCII-Code für %
$0809	(2057)	$28	(40)		PETSCII-Code für (
$080A	(2058)	$32	(50)		PETSCII-Code für 2
$080B	(2059)	$29	(41)		PETSCII-Code für)
$080C	(2060)	$00	(0)		Zeilenende
$080D	(2061)	$1B	(27)	L	Pointer auf 2075
$080E	(2062)	$08	(8)	H	*(27 + 8 x 256 = 2075)*
$080F	(2063)	$14	(20)	L	Zeilennummer 20
$0810	(2064)	$00	(0)	H	*(20 + 0 x 256 = 20)*
$0811	(2065)	$81	(129)		Token für FOR
$0812	(2066)	$20	(32)		PETSCII für SPACE
$0813	(2067)	$49	(73)		PETSCII für I
$0814	(2068)	$B2	(178)		Token für =
$0815	(2069)	$30	(48)		PETSCII für 0
$0816	(2070)	$20	(32)		PETSCII für SPACE
$0817	(2071)	$A4	(164)		Token für TO
$0818	(2072)	$20	(32)		PETSCII für SPACE
$0819	(2073)	$32	(50)		PETSCII für 2
$081A	(2074)	$00	(0)		Zeilenende
$081B	(2075)	$27	(39)	L	Pointer auf 2087
$081C	(2076)	$08	(8)	H	*(39 + 8 x 256 = 2087)*
$081D	(2077)	$1E	(30)	L	Zeilennummer 30
$081E	(2078)	$00	(0)	H	*(30 + 0 x 256 = 30)*
$081F	(2079)	$85	(133)		Token für INPUT
$0820	(2080)	$20	(32)		PETSCII für SPACE
$0821	(2081)	$41	(65)		PETSCII-Code für A
$0822	(2082)	$25	(37)		PETSCII-Code für %
$0823	(2083)	$28	(40)		PETSCII-Code für (
$0824	(2084)	$49	(73)		PETSCII-Code für I
$0825	(2085)	$29	(41)		PETSCII-Code für)
$0826	(2086)	$00	(0)		Zeilenende
$0827	(2087)	$2F	(47)	L	Pointer auf 2095
$0828	(2088)	$08	(8)	H	*(47 + 8 x 256 = 2095)*
$0829	(2089)	$28	(40)	L	Zeilennummer 40
$082A	(2090)	$00	(0)	H	*(40 + 0 x 256 = 40)*
$082B	(2091)	$82	(130)		Token für NEXT
$082C	(2092)	$20	(32)		PETSCII für SPACE
$082D	(2093)	$49	(73)		PETSCII-Code für I
$082E	(2094)	$00	(0)		Zeilenende
$082F	(2095)	$00	(0)	L	Programmende
$0830	(2096)	$00	(0)	H	

Abbildung 302 - Das Programm zur Speicherung eines Arrays im Speicher

Nun ist es wieder spannend zu sehen, wo sich denn nach dem Start des Programms und nach der Eingabe der Werte, diese im Speicher zu finden sind. Folgende Werte habe ich eingegeben.

Abbildung 303 - Die Eingabe der drei INTEGER-Werte

An welcher Stelle, sprich Startadresse, könnten sich die Daten befinden. Mag das die Adresse *2096* in dezimaler Schreibweise sein, die unmittelbar hinter dem Programmende frei ist? Sehen wir nach.

```
?PEEK(47)+256*PEEK(48)
 2106

READY.
```

Abbildung 304 - Die Startadresse der Array-Daten

Aha, dem ist also nicht so! Warum ist das nicht der Fall? Wir haben es bei Array-Variablen nicht mit einfachen Variablen zu tun, die unmittelbar dem BASIC-Programm im Anschluss folgen. Werfen wir noch einmal einen Blick auf die schon genannten und wichtigen Pointer, die definieren, wo welche Datentypen im Speicher lokalisiert sind.

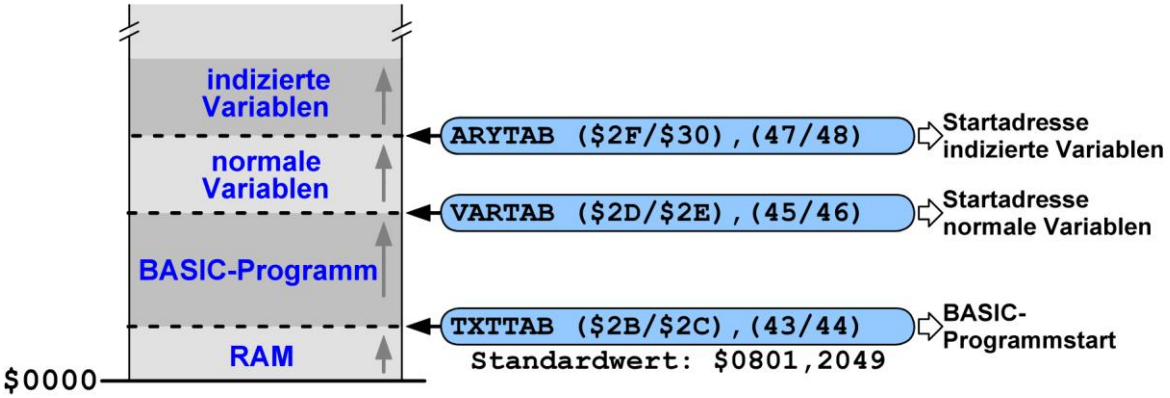


Abbildung 305 - Der ARYTAB-Pointer am oberen Rand

Es ist zu erkennen, dass der *ARYTAB*-Pointer auf den Bereich oberhalb des *VARTAB*-Pointers weist. Werfen wir also einen Blick in den Speicher ab Adresse *2106*, was die hexadezimale Adresse *$083A* ist.

```
.M083A 0849
:083A C1 80 0D 00 01 00 03 09   ▲.....
:0842 C2 FF 14 40 05 4D 4F 4E   |..@.MON
.
```

Abbildung 306 - Der Speicherbereich für die Array-Variable A

Bevor ich mit den Erläuterungen beginne, sollen wir einen Blick auf die Speicherung von Arrays ansehen.

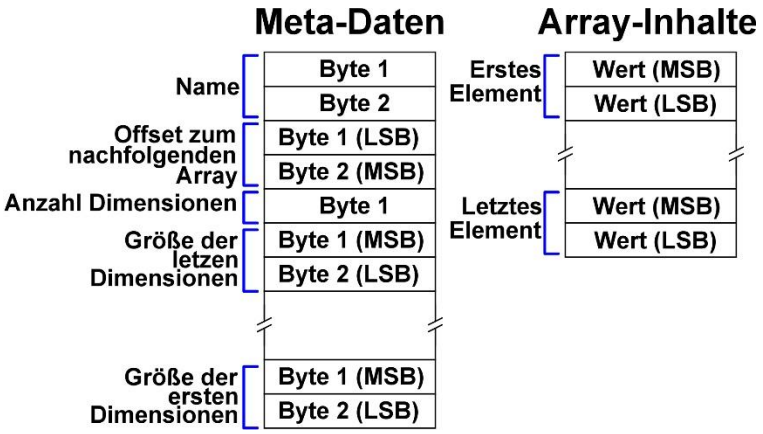


Abbildung 307 - Die Speicherung von Array-Elementen

Sehen wir uns schrittweise die Analyse der Bytes im Speicher an.

Der Name des Arrays

Der Name ist in den ersten beiden Bytes hinterlegt, der in unserem Fall wie folgt ausschaut und farblich hinterlegt ist.

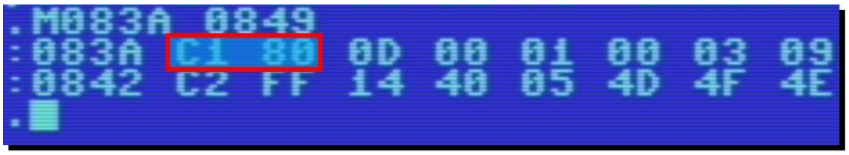


Abbildung 308 - Der Name des Arrays

Es sind die Werte *$C1* und *$80* zu sehen. Wir sollten uns das auf Bit-Ebene ansehen.

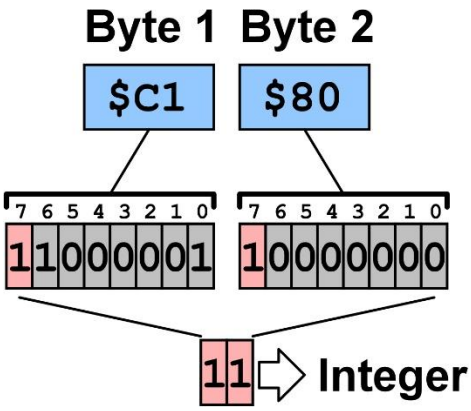


Abbildung 309 - Der Integer-Datentyp versteckt sich im Namen der Variablen

Um jetzt auf den eigentlichen Namen zu kommen, müssen die Bit-Kombinationen beider Bytes - wir kennen die Prozedur schon - wieder ohne Bit *7* gesehen werden, was für Byte *1* binär *1000001* bedeutet und den Wert *$41* repräsentiert. Hinsichtlich des PETSCII-Wertes erhält man wieder des Buchstaben *A*. Das gleiche wird mit Byte *2* gemacht, was binär den Wert *0000000* ergibt und besagt, dass die Variable lediglich aus einem Buchstaben besteht.

Der Offset-Wert zur nächsten Array-Variablen

Der Offset zeigt auf den Beginn der nächsten Array-Variablen über einen Offset-Wert. Dieser Wert ist hier farblich hervorgehoben, wobei das erste Byte das *LSB* repräsentiert.

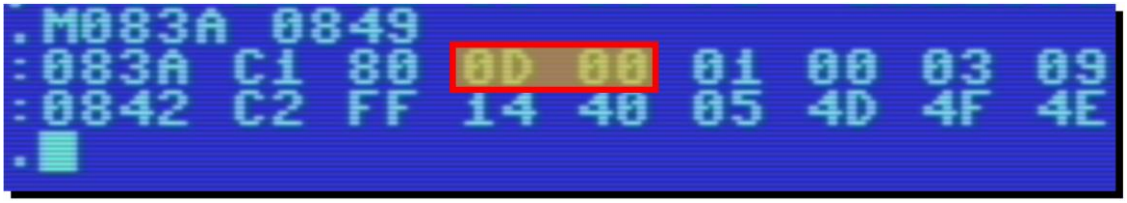


Abbildung 310 - Der Offset zum nächsten Array

Der Offset-Wert berechnet sich in dezimaler Schreibweise wie folgt.

$$Offset = 13 + 256 \cdot 0 = 13\ Bytes$$

Vom Beginn des Arrays im Speicher ab Speicherstelle *$083A* müssen also *13* Bytes hinzuaddiert werden, um zum Beginn des nächsten Arrays zu gelangen. Das MSB ist *$00* und braucht nicht berücksichtigt zu werden.

$$Startadresse = \$083A + \$0D = \$0847$$

Da wir im Moment jedoch lediglich ein einziges Array haben, spielt das hier keine wirkliche Rolle. Wir kommen im nächsten Beispiel mit zwei Arrays noch einmal darauf zu sprechen.

Die Anzahl der Dimensionen

Da ein Array aus mehreren Dimensionen bestehen kann, muss diese Information ebenfalls gespeichert werden, wozu ein einziges Byte ausreichend ist.

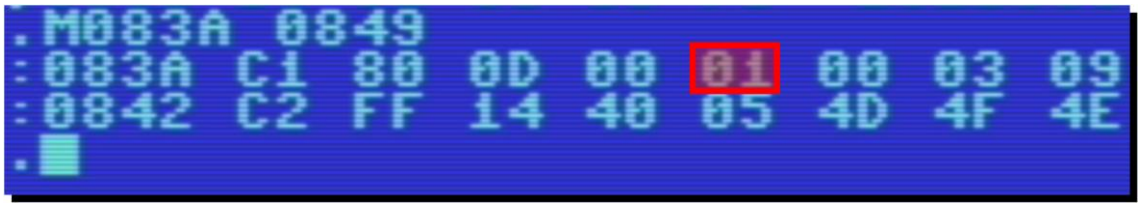


Abbildung 311 - Die Anzahl der Dimensionen

Da die Dimensionierung des Integer-Arrays in Zeile *10* eindimensional ist, beträgt der Wert hier natürlich *$01*.

10 DIM A%(2)

Das ist nicht mit der Anzahl der definierten Elemente zu verwechseln, die 3 (2 + 1) beträgt.

Die Größe der letzten Dimensionen

Wir haben es hier nur mit einer Dimension zu tun und deshalb wird auch nur dieses betrachtet. Die beiden nachfolgenden Bytes

sind wieder farblich hervorgehoben, wobei das erste Byte das *MSB* repräsentiert.

$$Größe\ des\ Arrays = 03 + 256 \cdot 0 = 03$$

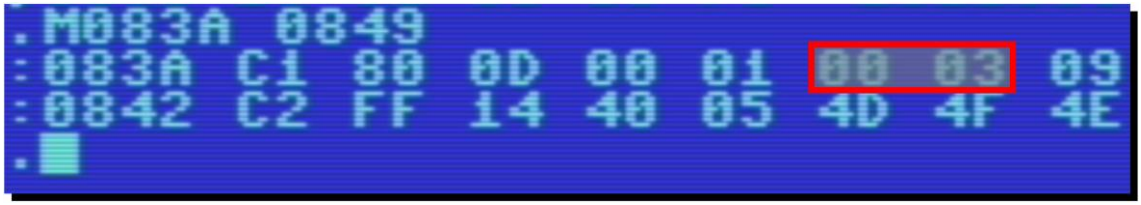


Abbildung 312 - Die Größe des letzten Arrays

Der Wert *3* gibt also genau an, dass wir es mit *3* Elementen zu tun haben, die jetzt unter die Lupe genommen werden.

Die Werte der einzelnen Array-Elemente

Nachfolgend habe ich die Bytes der einzelnen Array-Elemente farblich hervorgehoben, wobei jeder Integer-Wert natürlich aus zwei Bytes besteht und das *MSB* an erster Stelle steht.

Abbildung 313 - Die Werte der Array-Elemente

Sehen wir uns da wieder im Detail und beim zweiten Wert auf Bit-Ebene an. Es ist ja bekannt, dass es sich um Vorzeichen basierte Werte handelt und wenn das höchste Bit des MSB 1 ist, es sich um einen negativen Wert handelt. Es ist auf den ersten Blick zu erkennen, dass es sich lediglich beim zweiten Wert (*$FF / $14*) um einen derartigen Wert handelt. Die beiden anderen sind positiv. Von mir wurden nach dem Start des Programms die folgenden Werte eingegeben.

- **2498**
- **-236**
- **16389**

Erster Wert

$C2 (194) + *$09* (09) · 256 = *$9C2* (2498)

Zweiter Wert

Für den zweiten negativen Wert sollten wir einen Blick auf die Bit-Ebene werfen. Um den positiven Wert zu ermitteln, muss zuerst das Einer- und dann das Zweierkomplement gebildet werden, wie das auf der folgenden Abbildung zu erkennen ist.

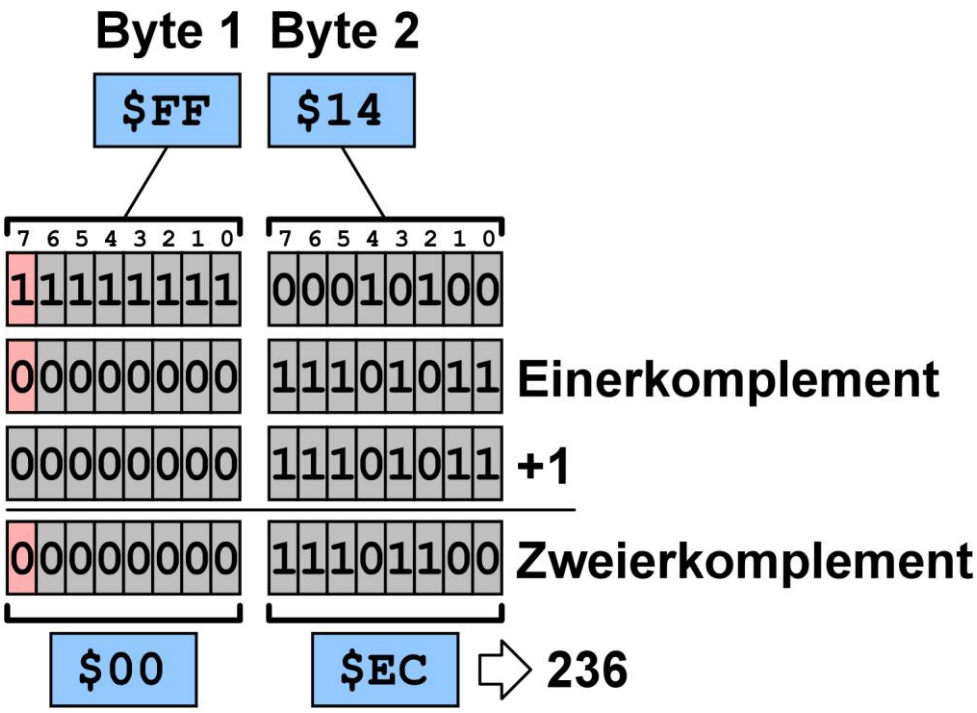


Abbildung 314 - Die Ermittlung des positiven Wertes

Dritter Wert

$05 (05) + *$40* (64) · 256 = *$4005* (16389)

Ein erweitertes Array-Beispiel

Wie versprochen, möchte ich jetzt ein etwas erweitertes Beispiel zur Speicherung von Array-Variablen zeigen. Ich mache das der Einfachheit wegen wieder mit Integer-Werten, doch jeder sollte selbst einmal die anderen Datentypen durchspielen. Folgendes Programm liegt vor.

```
10 DIM C%(2)
20 DIM D%(3)
30 C%(0)=1:C%(1)=3:C%(2)=5
40 D%(0)=2:D%(1)=4:D%(2)=6:D%(3)=8
RUN

READY.
```

Abbildung 315 - Das Programm zur Speicherung zweier Arrays

Das eigentliche BASIC-Programm werde ich diesmal nicht im Speicher untersuchen und direkt auf den Bereich wechseln, in dem die Array-Variablen ihre Werte ablegen. Natürlich ist es zuvor wieder wichtig zu wissen, worauf der *ARYTAB*-Pointer zeigt.

Abbildung 316 - Die Ermittlung des ARYTAB-Pointers

Der dezimale Wert *2139* entspricht dem hexadezimalen Wert *$085B*. Sehen wir also wieder in SMON nach und gehen schrittweise vor.

Der Name der Arrays

Die Namen der beiden Array *C%* und *D%* sehen wir auf der folgenden Abbildung farblich hervorgehoben, wobei ich natürlich den Offset der ersten Variablen zur Trennung genutzt habe, der wieder *13* (*$0D*) Bytes beträgt und vom *LSB* des ersten Variablennamen gezählt wird.

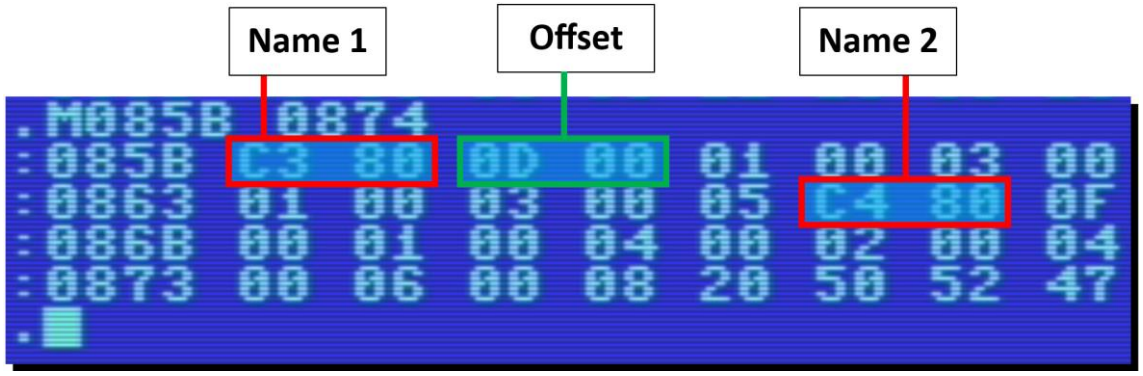


Abbildung 317 - Die Namen und der erste Offset

Der beiden Array-Namen lauten *C%* und *D%* und sind durch *$C3 / $80* und *$C4 / $80* definiert. Wir erinnern uns, dass von jedem Byte das höchstwertige Bit ignoriert werden muss, so dass sich die PETSCII-Codes *$43* (*67*) und *$44* (*68*) ergeben.

Die Offset-Werte zu den nächsten Array-Variablen

Da wir es hier mit zwei Array-Variablen zu tun haben, sind auch zwei Offset-Werte vorhanden.

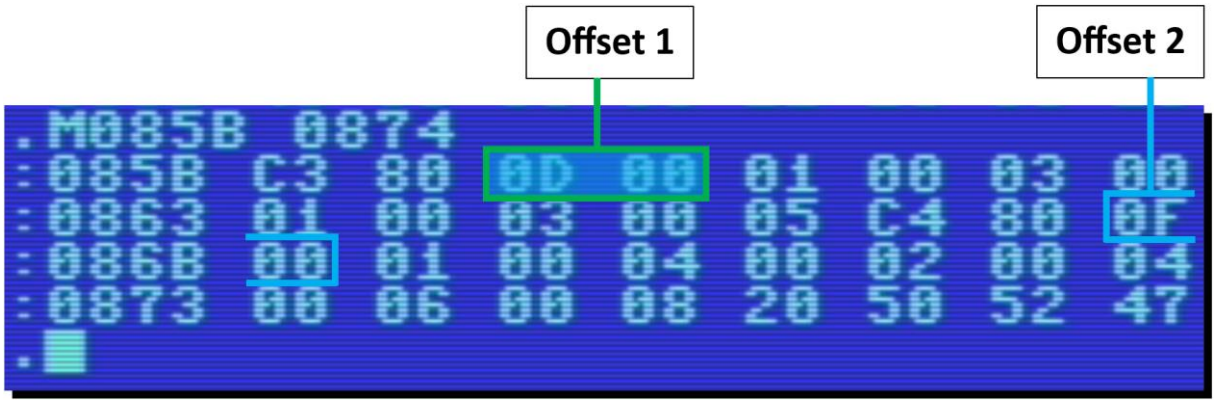


Abbildung 318 - Die beiden Offset-Werte

Der erste Offset-Wert - wir hatten es schon gesehen - beträgt *$0D* (*13*), was besagt, dass das zweite Array *13* Bytes später als das erste Array beginnt. Das *MSB* ist *$00* und braucht wieder nicht berücksichtigt zu werden.

Vom Beginn des ersten Arrays *C%* im Speicher ab Speicherstelle *$085B* müssen demnach *13* Bytes hinzuaddiert werden, um zum Beginn des nächsten Arrays *D%* zu gelangen.

$$\boldsymbol{Startadresse = \$085B + \$0D = \$0868}$$

Diese Adresse weist genau auf das Byte, an dem *$C4* hinterlegt ist und den Namen von *D%* vergibt. Da das zweite Array mit seinen *4* Elemente größer ist, als das erste, ist auch der Offset-Wert auf das mögliche nächste Array größer und lautet *$0F*, was *15* Bytes entspricht. Die mögliche Startadresse würde demnach wie folgt berechnet.

$$Startadresse = \$0868 + \$0F = \$0877$$

Die Anzahl der Dimensionen

Da die beiden Arrays nur eine einzige Dimension vorweisen, steht auch an den betreffenden Positionen der Wert *$01*.

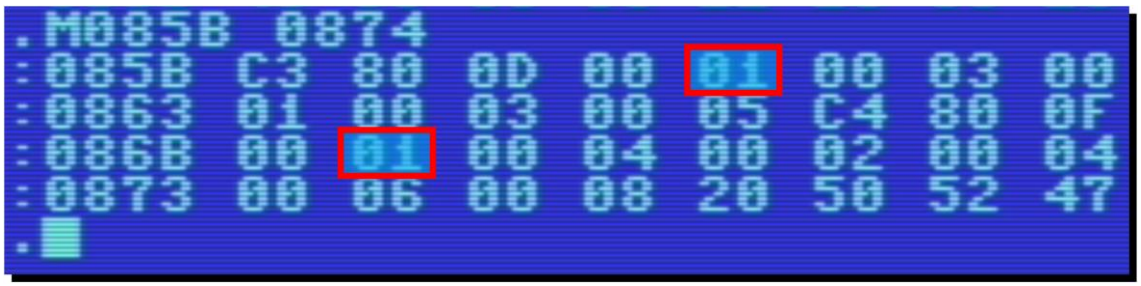


Abbildung 319 - Die Anzahl der Dimensionen

Die Größe der letzten Dimensionen

Nun besitzt das erste Array *3* Elemente und das zweite *4* Elemente, was an den markierten Positionen zu sehen ist. Bei den beiden Bytes je Pointer wird das *MSB* zuerst genannt.

Abbildung 320 - Die Anzahl der Dimensionen

Die Werte der einzelnen Array-Elemente

Nachfolgend habe ich die Bytes der einzelnen Array-Elemente je Array wieder farblich hervorgehoben. Die Elemente des *C%*-Arrays sind in Rot, die des *D%*-Elementes in Blau hervorgehoben.

```
.M085B 0874
:085B C3 80 0D 00 01 00 03 00
:0863 01 00 03 00 05 C4 80 0F
:086B 00 01 00 04 00 02 00 04
:0873 00 06 00 08 20 50 52 47
.
```

Abbildung 321 - Die einzelnen Array-Elemente der beiden Arrays

Wir haben gesehen, wie Integer-Variablen mit ihren Werten im Speicher organisiert sind. Es fehlen noch die Informationen über Fließkomma-Zahlen und Zeichenketten. Die Struktur der Meta-Daten wie Name, Offset, Anzahl Dimensionen, etc. ist für alle immer gleich und ändert sich nicht. Nachfolgend ist die Grafik für alle drei einmal zu sehen.

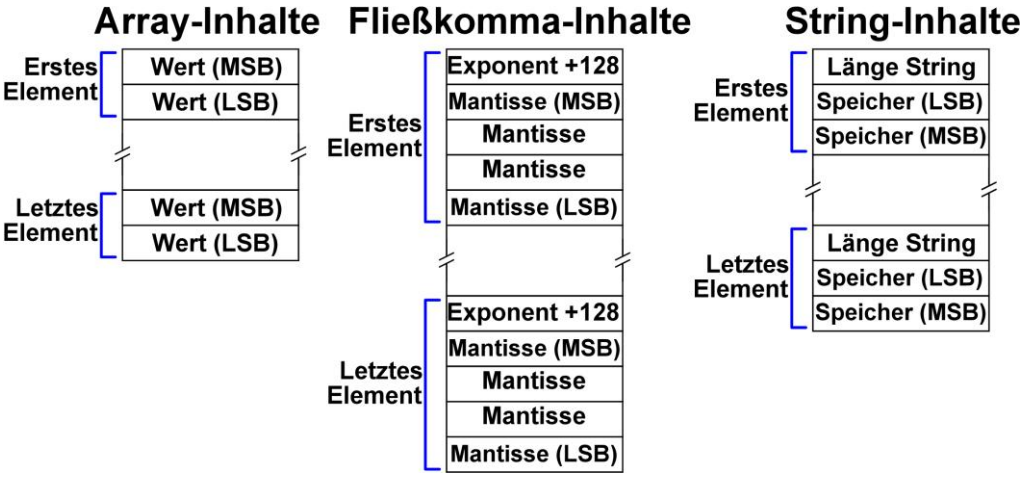


Abbildung 322 - Die Speicherung der Daten der unterschiedlichen Datentypen

Eine Funktion

Die Daten für eine Funktionen werden in fünf Bytes gespeichert. Die ersten beiden Bytes dienen als Pointer auf den Körper der Funktionsdefinition innerhalb des Programms unmittelbar hinter dem "="-Zeichen der *DEF-FN*-Definition. Die darauffolgenden beiden Bytes enthalten die Adresse des Datenfeldes für die Variable, die das Argument der Funktion darstellt. Das letzte Byte enthält das erste Byte der Funktionsdefinition. Nachfolgend ist die Darstellung der einzelnen Bytes für eine Funktion zu sehen.

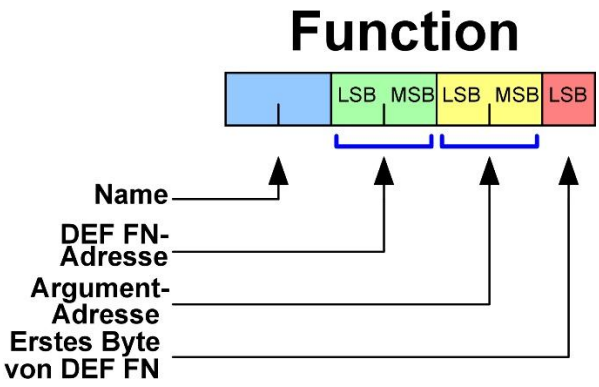


Abbildung 323 - Die 5 Bytes des Function-Datentyps

Sehen wir uns dazu wieder ein kleines Beispiel an. Wir haben ja gesehen, dass es über den *PEEK*-Befehl möglich ist, auf Speicherinhalte zuzugreifen und schon sehr oft davon Gebrauch gemacht, wenn es um die Adress-Ermittlung bei Zeigern geht. Allgemein lautet das ja.

$$Adresse = PEEK(LSB) + 256 \cdot PEEK(MSB)$$

Um die Sache innerhalb eines Programms zu vereinfachen, kann man - ich hatte das Beispiel schon gebracht - sich eine Funktion programmieren, die diese Aufgabe übernimmt. Das könnte wie folgt aussehen.

```
10 DEF FN AD(X)=PEEK(X)+256*PEEK(X+1)
20 INPUT X
30 PRINT FN AD(X)
READY.
```

Abbildung 324 - Das Programm zur Ermittlung der Adresse eines Zeigers

Funktionen werden von in BASIC als einfache Variablen behandelt. Somit ist die Kennzeichnung im Zwei-Byte-Namen beim ersten Byte mit gesetztem Bit 7 und beim zweiten Byte mit gelöschtem Bit 7. Unser Funktions-Name *AD* würde also mit *$C1* und *$44* im Speicher stehen, aber im Endeffekt auf die Werte *$41* und *$44* deuten. Sehen wir also im Speicher nach und nutzen direkt das Programm zur Ermittlung der Adresse über den Pointer *VARTAB*.

```
? 45
 2103

READY.
```

Abbildung 325 - Die Ermittlung der VARTAB-Adresse

Die dezimale Adresse *2103* lautet in hexadezimaler Schreibweise *$0837* und damit rufen wir die Adresse in SMON auf. Sehen wir uns wieder die Details an.

Der Name des Arrays

Der Name ist in den ersten beiden Bytes hinterlegt, der in unserem Fall wie folgt ausschaut und farblich hinterlegt ist.

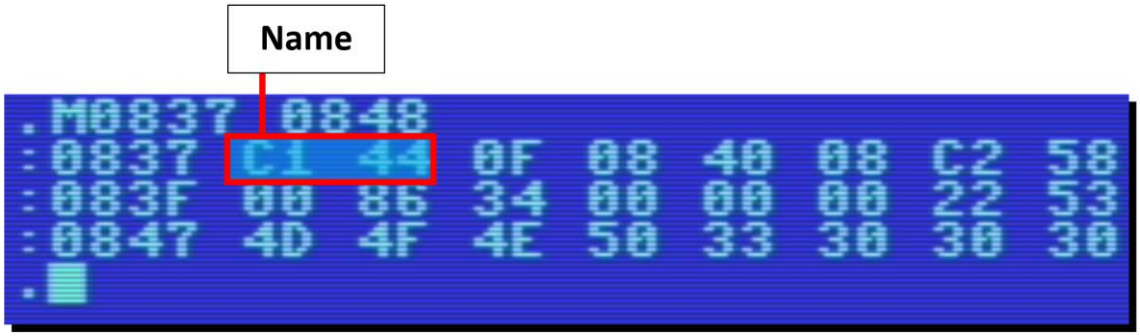


Abbildung 326 - Der Funktions-Name

Die DEF FN-Adresse

Die *DEF FN*-Adresse ergibt sich aus den folgenden zwei Bytes, wobei das *LSB* an erster Stelle steht. Dieser Zeiger weist auf den Körper der Funktionsdefinition innerhalb des Programms, also auf den Teil nach dem "="-Zeichen in der der *DEF-FN*-Definition.

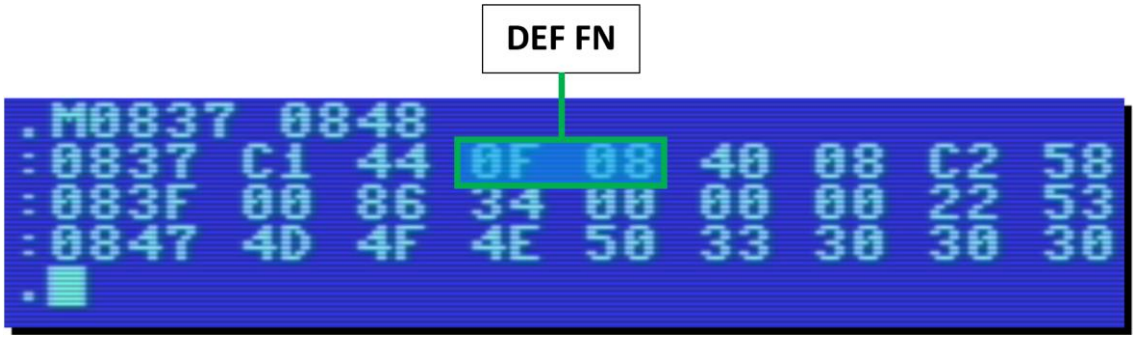


Abbildung 327 - Der Pointer auf die DEF-FN-Definition

In unserem Fall lautet die Adresse des Zeigers

$$Adresse = \$0F + 256 \cdot \$08 = \$080F\ (2063)$$

Wir sehen uns gleich diesen Speicher ab Adresse *$080F* genauer an.

Die Dummy-Argument-Adresse

Die nächsten beiden Bytes enthalten die Adresse für die Variable, die das Argument der Funktion darstellt.

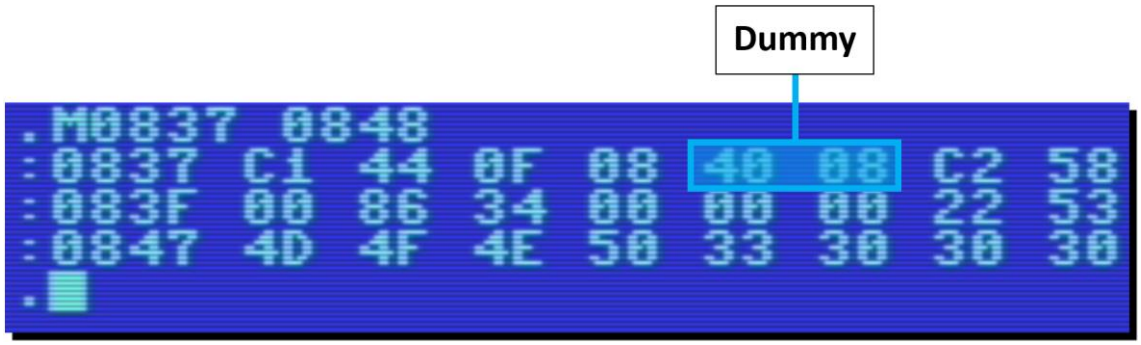


Abbildung 328 - Die Adresse der Argument-Variablen

Das erste Byte von DEF FN

Im letzten Byte ist das erste Byte der Funktions-Definition gespeichert, was wir bei der übernächsten Abbildung bei der Detailauflistung sehen. Dort besitzt das erste Byte genau den Wert $C2.

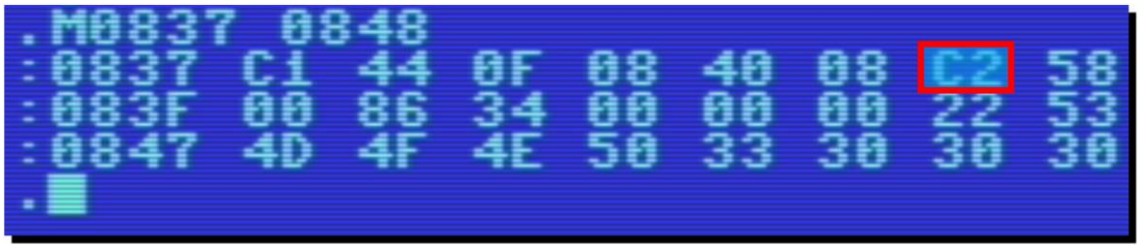


Abbildung 329 - Das erste byte der DEF-FN-Definition

Wie erwähnt, sehen wir uns jetzt den Speicher ab Adresse *$080F* an, worauf ja der Zeiger für die DEF FN-Definition weist.

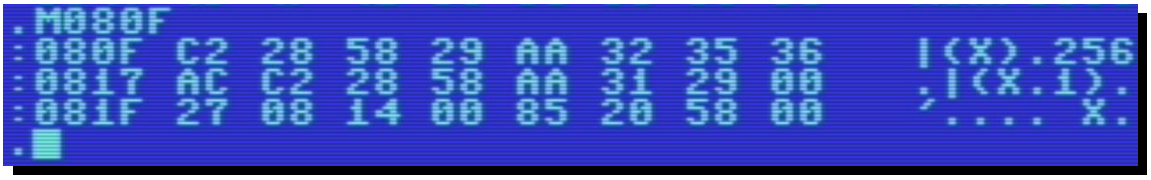


Abbildung 330 - Die DEF FN-Definition im Speicher

Sehen wir uns die Aufschlüsselung dazu an.

Adresse HEX	Adresse Dez	Inhalt HEX	Inhalt Dez	Erläuterungen
$080F	(2063)	$C2	(194)	Token für PEEK
$0810	(2064)	$28	(40)	PETSCII-Code für (
$0811	(2065)	$58	(88)	PETSCII-Code für X
$0812	(2066)	$29	(41)	PETSCII-Code für)
$0813	(2067)	$AA	(170)	Token für +
$0814	(2068)	$32	(50)	PETSCII-Code für 2
$0815	(2069)	$35	(53)	PETSCII-Code für 5
$0816	(2070)	$36	(54)	PETSCII-Code für 6
$0817	(2071)	$AC	(172)	Token für *
$0818	(2072)	$C2	(194)	Token für PEEK
$0819	(2073)	$28	(40)	PETSCII-Code für (
$081A	(2074)	$58	(88)	PETSCII-Code für X
$081B	(2075)	$AA	(170)	Token für +
$081C	(2076)	$31	(49)	PETSCII-Code für 1
$081D	(2077)	$29	(41)	PETSCII-Code für)
$081E	(2078)	$00	(0)	Zeilenende

Abbildung 331 - Die Entschlüsselung der Funktions-Definition

Sehen wir uns abschließend noch die Auflistung der genannten Datentypen untereinander an.

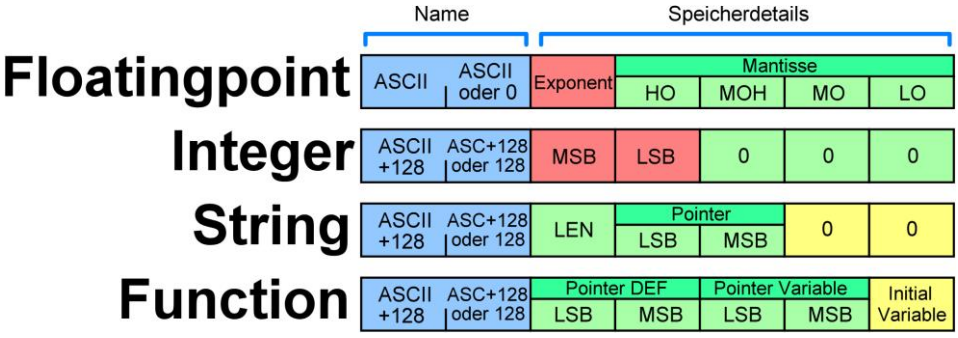


Abbildung 332 - Die einfachen Datentypen im Speicher

Und nun die Speicherung von Arrays.

Abbildung 333 - Arrays im Speicher

Die Tastatur

- Der CIA-Baustein
- Die Portregister
- Die Datenrichtungsregister
- Die Tastaturabfrage

Die Tastatur - Das Keyboard

Wenn es darum geht, die *66* Tasten auf dem Keyboard des C64 abzufragen, dann geht das in einer besonderen Weise. Man könnte natürlich jede einzelne Taste mit Kabeln versehen und diese dann einzeln abfragen, was einen nicht unerheblichen Aufwand bedeuten würde. Aus diesem Grund sind die *64* Tasten in einer *8x8*-Matrix organisiert und werden in einer bestimmten Weise abgefragt.

Die Aufgabe übernimmt ein sogenannter *CIA*-Baustein. CIA steht für ***C****omplex* ***I****nterface* ***A****dapter*. Dieser Adapter (CIA oder *6526*) ist ein *40*-Pin-Chip, der für die Verwaltung von Schnittstellen entwickelt wurde. Er ist ein Nachfahre der *PIA*- und *VIA*-Chips in anderen Commodore-Maschinen und ist einfacher zu programmieren.

Der CIA-Baustein

Der C64 enthält zwei CIAs, die große Ähnlichkeit zu den *6522* VIAs aufweisen. Eingabegeräte (Tastaturen, Taster und ähnliches) oder Ausgabegeräte (wie LEDs) können über einen Puffer direkt mit dem Chip verbunden und direkt über eine Speicherstelle angesprochen werden. Man kann sagen, dass die CIAs den kompletten Verkehr mit der Außenwelt des C64 steuern. Diese Chips besitzen zudem eine interessante Fähigkeit wie die zwei eingebauten *16*-Bit Timer und eine Echtzeituhr mit Alarm. Dieser Baustein besitzt, ähnlich wie die CPU, ebenfalls Register, die zur internen Konfiguration genutzt werden. Es handelt sich also um Speicherstellen, die vor der Nutzung mit bestimmten Werten befüllt sein müssen. Ich komme gleich dazu.

Um eine *8x8*-Matrix realisieren zu können, müssen *16* Pins zur Verfügung stehen. Genau das bietet der CIA-Baustein mit seinen zwei sogenannten Ports mit je *8* Leitungen. Werfen wir dazu einen kurzen Blick auf den Baustein, ohne tiefer auf andere Pins einzugehen.

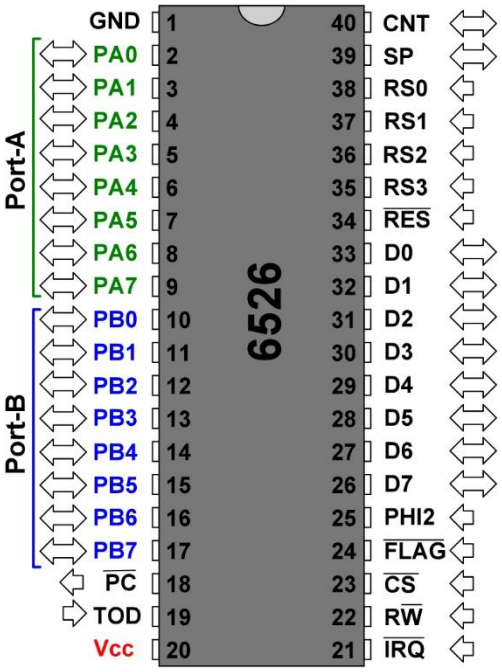


Abbildung 334 - Die Pinbelegung des CIA-Bausteins 6526

Auf der linken Seite sind die beiden *8*-Bit Ports *PA* und *PB* zu sehen, an denen die Tastatur des C64 angeschlossen ist. Der C64 besitzt übrigens zwei (*U1* und *U2*) dieser CIA-Bausteine. Ihre Adressen sind

- ***CIA1 - U1***: *$DC00* bis *$DC0F* - (*56320-56335*)
- ***CIA2 - U2***: *$DD00* bis *$DD0F* - (*56576-56591*)

Konzentrieren wir uns aber auf den *CIA1*, der unter anderem für die Abfrage des Keyboards zuständig ist. Zwei Register sind dafür von Bedeutung. Nähere Informationen zum *CIA* sind unter den folgenden Internetadressen zu finden.

Die Funktionsweise eines CIA

- https://www.c64-wiki.com/wiki/CIA
- http://cbmmuseum.kuto.de/zusatz_6526_cia.html

Die Portregister

Fangen wir mit den beiden Portregistern an, die quasi die Sensoren zur Außenwelt darstellen. Jedes dieser Register besitzt natürlich *8* Bits, die es zu steuern beziehungsweise auszuwerten gilt. Diese Ports sind jedoch sehr flexibel, so dass jedes dieser Pins einzeln konfigurierbar ist. Also nicht nur der gesamte Port, sondern jede einzelne Leitung. Die gewünschte Funktion, die erforderlich ist, kann softwaremäßig festgelegt werden. Über das Setzen oder Löschen der Bits können die Pins entweder als Ein-

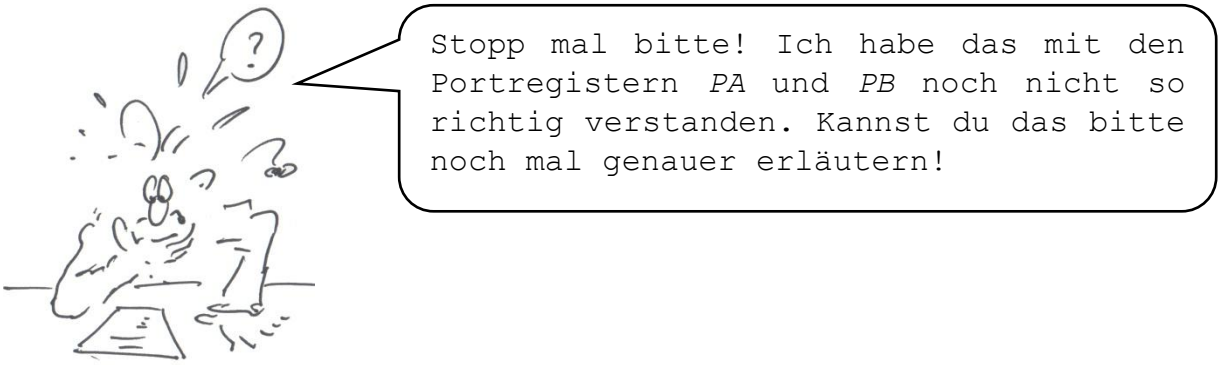
oder Ausgänge eingesetzt werden. Wir müssen nun ein wenig mit den Adressen hantieren, um die Funktionsweise zu verstehen.

- ***Port A-Register*** also *PA* besitzt die Adresse *$DC00 (56320)*
- ***Port B-Register*** also *PB* besitzt die Adresse *$DC01 (56321)*

Für die Programmierung werden in der Regel standard Abkürzungen verwendet, die in den Code-Listings zu finden sind. In der nachfolgenden Tabelle habe ich diese aufgelistet.

Registernummer	Abkürzung	Name
0	PRA	Portregister A
1	PRB	Portregister B

Tabelle 46 - Die Portregister



Klar, das ist kein Problem. Nehmen wir zum Beispiel den *Port A* mit der Kurzbezeichnung *PA*. Es handelt sich dabei um acht Datenleitungen, die ähnlich wie bei der *6502-/6510*-CPU der Datenbus, mit bestimmten Werten versehen werden kann. Es ist die Schnittstelle nach draußen. Auf der nachfolgenden Abbildung habe ich an den Pins des *Portregisters A* einfach ein paar Lampen angeschlossen, was man in der Realität natürlich so nicht machen darf. Das müssten dann Leuchtdioden mit Vorwiderständen sein. Doch so ist es ein wenig eindrücklicher.

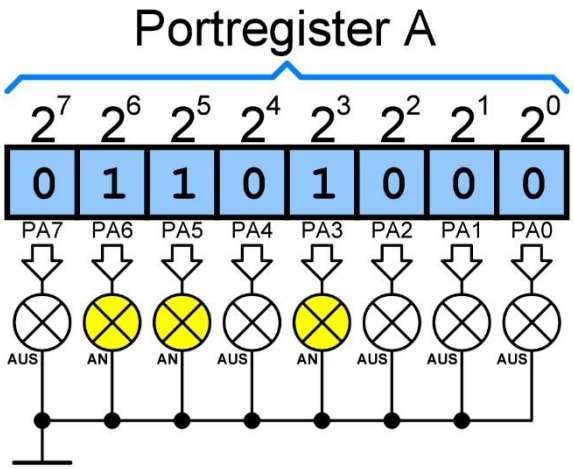


Abbildung 335 - Die Bits des Portregisters A (alles Ausgänge)

Alle Pins des Ports arbeiten als Ausgänge und der Strom fließt also - technisch gesehen - aus den Pins heraus und durch die

angesteuerten Lampen. Natürlich können auch Taster (Tasten) als Eingabeelemente individuell angeschlossen werden. Wenn das der Fall ist, muss der jeweilige Pin umkonfiguriert werden, was mich zum nächsten wichtigen Punkt bringt. Das Datenrichtungsregister!

Die Datenrichtungsregister

Nun haben wir gesehen, welches Register Verbindungen zur Außenwelt herstellen. Es gilt also jetzt zu definieren, welche der Pins als Ein- und welche als Ausgänge arbeiten sollen. Die Konfiguration der Ports mit den entsprechenden Pins hinsichtlich der Datenflussrichtung wird über ein weiteres Register vorgenommen, das sich *DDR* (***D**ata-**D**irection-**R**egister*) nennt. Es wird auch *Datenrichtungsregister* genannt.

Ist ein einzelnes Bit im Datenrichtungsregister eines Ports gelöscht, also mit einer *0* versehen, so ist der entsprechende Pin des Ports als Eingang konfiguriert; ist es gesetzt, also mit einer *1* versehen, so ist der Pin des Ports als Ausgang anzusehen.

Bitwert des DDR	Konfiguration
0	Eingang
1	Ausgang

Tabelle 47 - Die Konfiguration des DDR

Sollen also zum Beispiel alle Pins eines Ports als Ausgänge (OUTPUT) arbeiten, muss der hexadezimale Wert *$FF* (binär *11111111*) im entsprechenden DDR hinterlegt sein. Etwas sehr Wichtiges muss bei der Programmierung jedoch beachtet werden. Im Umgang mit den Portregister müssen wir auf jeden Fall darauf achten, dass im CIA ein *Inverter* - Logikumkehrer - eingebaut ist der die Ein- und Ausgangssignale ins Gegenteil wandelt. Warum das so ist? Nächste Frage bitte!

Wir müssen natürlich auch hier ein wenig mit den Adressen hantieren, um die Funktionsweise zu verstehen.

- ***DDRA*** besitzt die Adresse *$DC02 (56322)*
- ***DDRB*** besitzt die Adresse *$DC03 (56323)*

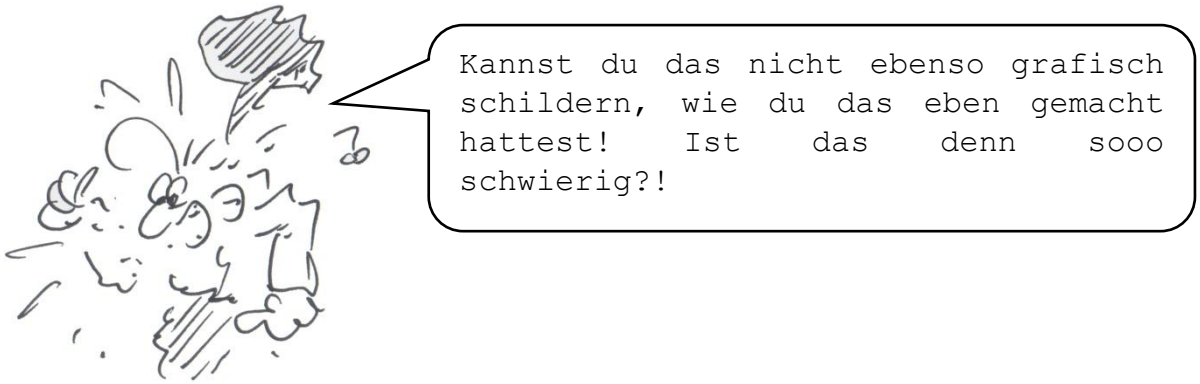
Für die Programmierung werden in der Regel standard Abkürzungen verwendet, die in den Code-Listings zu finden sind. In der nachfolgenden Tabelle habe ich diese aufgelistet.

Registernummer	Abkürzung	Name
2	DDRA	Data Direction Register A
3	DDRB	Data Direction Register B

Tabelle 48 - Die Data-Direction-Register

Es muss beim Lesen eines Ports darauf geachtet werden, dass die Eingangsbits immer in invertierter Schreibweise im Register erscheinen. Ist zum Beispiel ein Port als Eingang konfiguriert und es liegt ein *Low*-Pegel (*0*: kein Signal) an einem der Pins an, dann sind die entsprechenden Bits im Datenportregister auf

1. Liegt ein *High*-Pegel (*1*: Signal vorhanden) an einem der Bits an, so erscheint im Portregister dies als eine *0*.



Bleib locker! Angenommen, ich möchte an einem Port Ausgänge und Eingänge gemischt nutzen, wie das auf der folgenden Abbildung zu erkennen ist. Die ***x*** stehen hier stellvertretend für einen noch nicht definierten Zustand, der natürlich vom Tasterstatus abhängt.

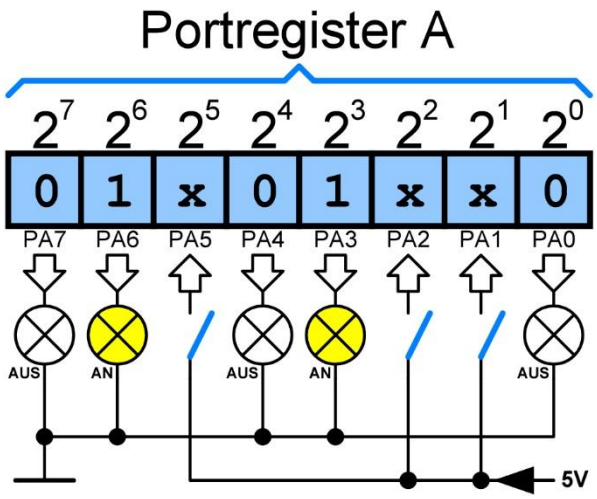


Abbildung 336 - Die Bits des Portregisters A (Ein- und Ausgänge)

In diesem Fall muss für jeden Pin das entsprechende Bit gesetzt oder gelöscht werden. Wir erinnern uns: Eine *1* steht für einen Ausgang und eine *0* für einen Eingang. Das *DDRA* würde demnach wie folgt gesetzt werden müssen.

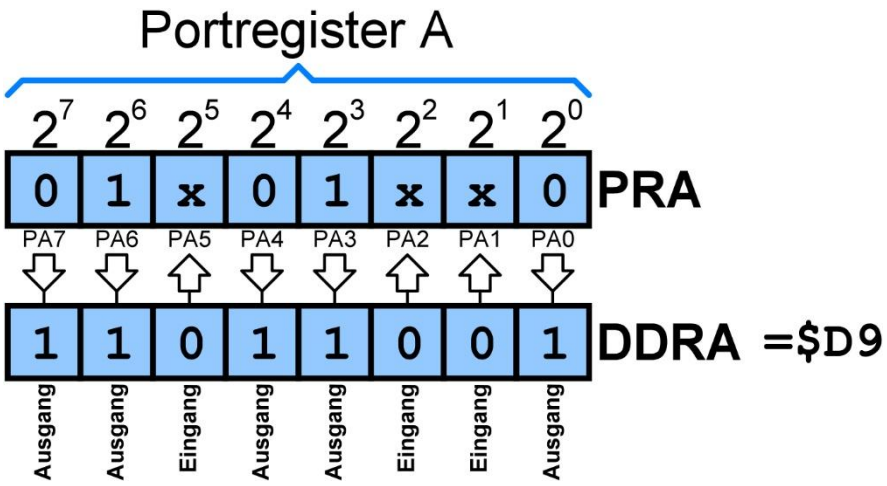


Abbildung 337 – Das DDRA konfiguriert das Portregister A

Die Tastaturabfrage

Kommen wir nun zur Tastaturabfrage am C64. Die Tastatur wird mit Hilfe der Portregister *A* und *B* der *CIA1* (U1) abgefragt. Eigentlich sollte man meinen, dass auf diese Weise mit den zur Verfügung stehenden *16* Pins der beiden Ports *A* und *B* lediglich *16* Tasten abgefragt werden können. Man kann jedoch mit *2x8* Bits eine Matrix bilden, mit der $2^8=64$ Kombinationen abgefragt werden können. Werfen wir einen Blick auf diese Matrix. Lasse dich nicht verunsichern, wenn im Internet anders angeordnete Grafiken zu sehen sind. Da können Port *A* und *B* vertauscht sein oder auch die einzelnen Bits in umgekehrter Richtung zu finden sein. Das spielt im Endeffekt keine Rolle, weil es nur eine andere Darstellungsweise ist. Hier muss man ein wenig flexibel sein.

C64-Tastatur Matrix		Port A CIA-1 A: Register $DC00							
		PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
Port B CIA-1 B: Register $DC01	PB0	1	£	+	9	7	5	3	<DEL>
	PB1	<←→>	*	P	I	Y	R	W	<RETURN>
	PB2	<CTRL>	;	L	J	G	D	A	<CRSR◄►>
	PB3	2	<HOME>	-	0	8	6	4	F7
	PB4	<SPACE>	<R.SHIFT>	.	M	B	C	Z	F1
	PB5	<C=>	=	:	K	H	F	S	F3
	PB6	Q	↑	@	O	U	T	E	F5
	PB7	<RUN/STOP>	/	,	N	V	X	<L.SHIFT>	<CRSR▲▼>

Abbildung 338 Die Matrix zur Tastaturabfrage

Man kann sich eine Keyboard-Matrix wie folgt vorstellen, wobei ich nur ein paar Tasten dargestellt habe. Es ist zu sehen, wie

die einzelnen Tasten über waagerechte und senkreckte Leitungen miteinander verbunden sind und sich an den Kreuzungspunkten befinden. Wird zum Beispiel eine Taste gedrückt, kommt es - je nach vorheriger Konfiguration der Ports - zu einem Stromfluss wie das auch bei einem der Taster (Zeile *1* / Spalte *1*) zu sehen ist.

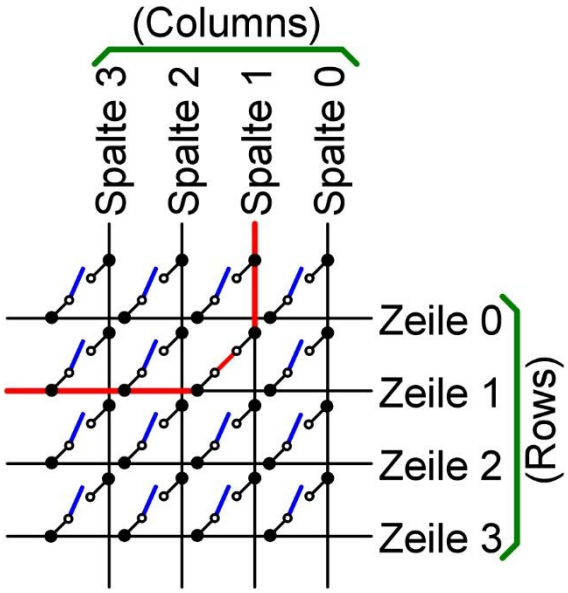


Abbildung 339 - Eine einfache Keyboard-Matrix

Hinsichtlich des Keyboards wird *PA* die Ansteuerung der Spalten (Columns) und *PB* die Ansteuerung der Reihen (Rows) übernehmen. Ich möchte nun konkret werden und ein Programm schreiben, das solange in einer Schleife hängt, bis die Taste ***S*** gedrückt wird. Ich habe dazu die Keyboard-Matrix auf den CIA-Baustein *gemappt*, wie man das neudeutsch so sagt. Das schaut dann wie folgt aus.

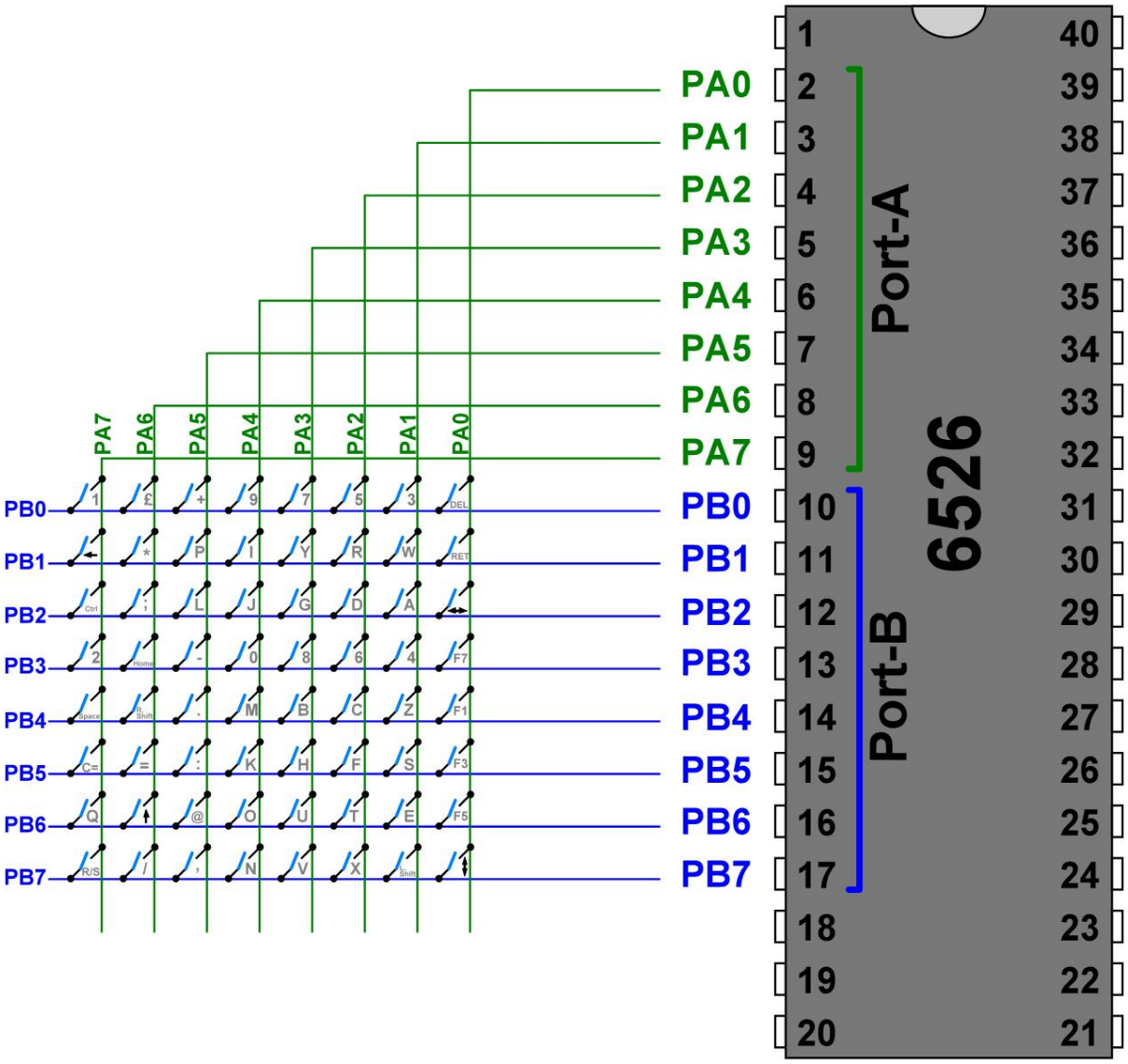


Abbildung 340 - Die Keyboard-Tasten sind mit dem CIA-Baustein verbunden

Ich hatte das hirnrissige Verhalten hinsichtlich der Signalpegel für Eingangssignale schon kurz erwähnt. Was aus der Grafik nicht ersichtlich ist, ist der Umstand, dass alle nicht gedrückten Tasten einen *High*-Pegel aufweisen, so dass sie also eine *1* bei der Abfrage liefern! Um zu erkennen, ob eine Taste gedrückt wurde, muss auf einen Low-Pegel hin geprüft werden, was also der *0* entspricht. Das macht doch wirklich Spaß, oder?!

Doch weiter im Text. Zuerst muss der CIA konfiguriert werden. Dazu wird *Port A* komplett als Ausgang und *Port B* komplett als Eingang gesetzt. Wir erinnern uns, dass ein Signal-Pin als Ausgang arbeitet, wenn im DDR eine *1* an der betreffenden Bitposition steht. Umgekehrt muss ein Signal-Pin, der Eingangssignale entgegennehmen will, im DDR eine *0* an der betreffenden Bitposition vorweisen. Demnach muss *Port A* mit dem Wert *$FF* im *DDRA* versorgt werden und *Port B* mit dem Wert *$00* im DDRB, was dann auf der folgenden Abbildung zu sehen ist.

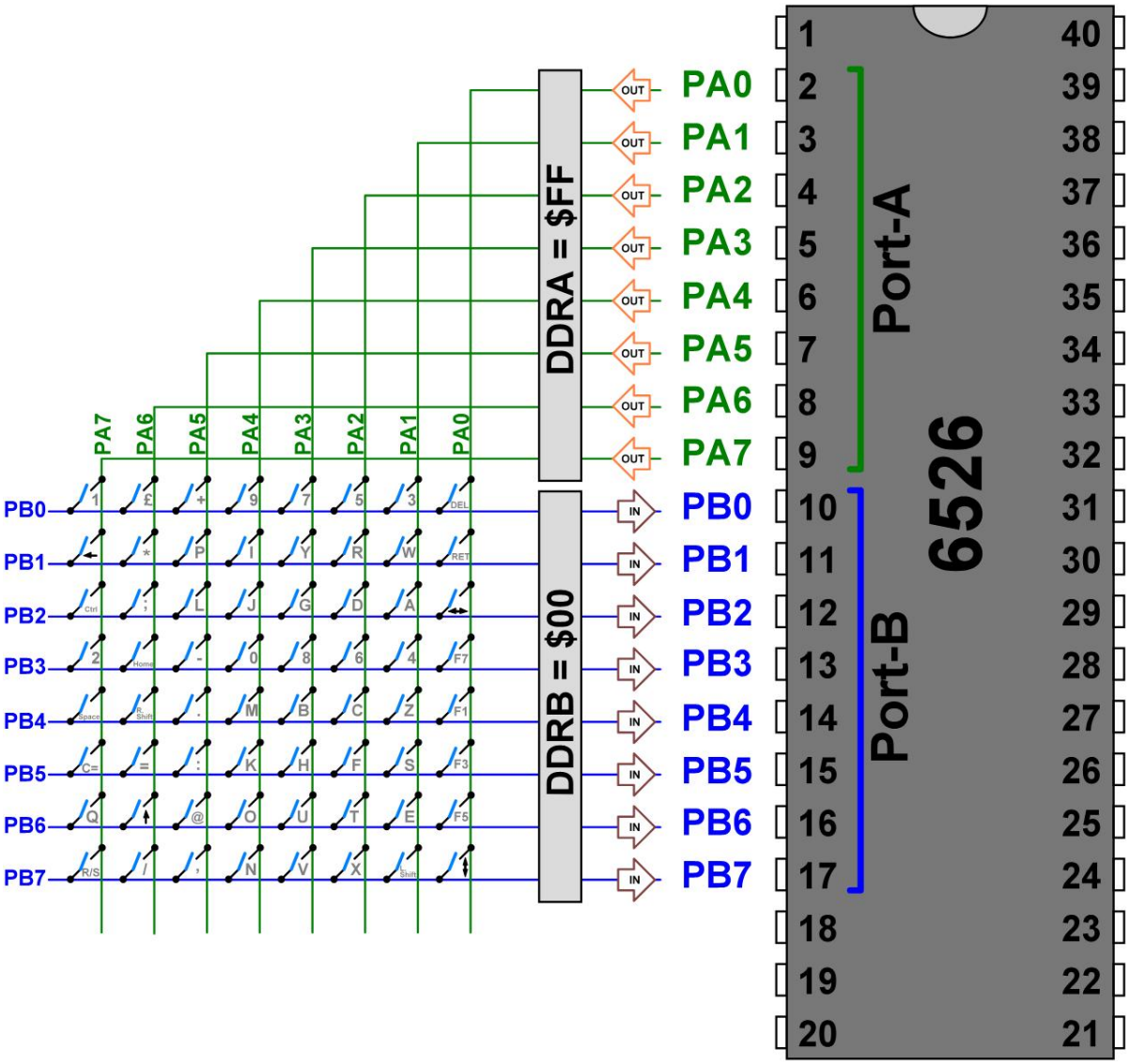


Abbildung 341 - Die Konfiguration Datenflussrichtung der der Ports

In der Programmierung schaut das dann wie folgt aus, wobei ich schrittweise vorgehe und den Code etwas auseinanderziehe. Den ersten Befehl an Adresse *$4000* überspringen wir hier. Der *SEI*-Befehl setzt das Interrupt-Flag. Dadurch wird die Interrupt-Sperre auf *1* gesetzt. Sie wird verwendet, um Interrupt-Anforderungen während System-Reset-Operationen und während Interrupt-Befehlen zu maskieren also in diesem Fall zu deaktivieren, damit eine eventuell auftretende Interrupt-Anforderung nicht störend auf dieses Programm wirkt.

Teil 1: Initialisierung von DDRA

Im ersten Teil wird das *DDRA* mit dem Wert *$FF* versehen, so dass alle Pins als Ausgänge arbeiten.

Abbildung 342 - Programm zur Abfrage der Taste S - Teil 1

Teil 2: Initialisierung von DDRB

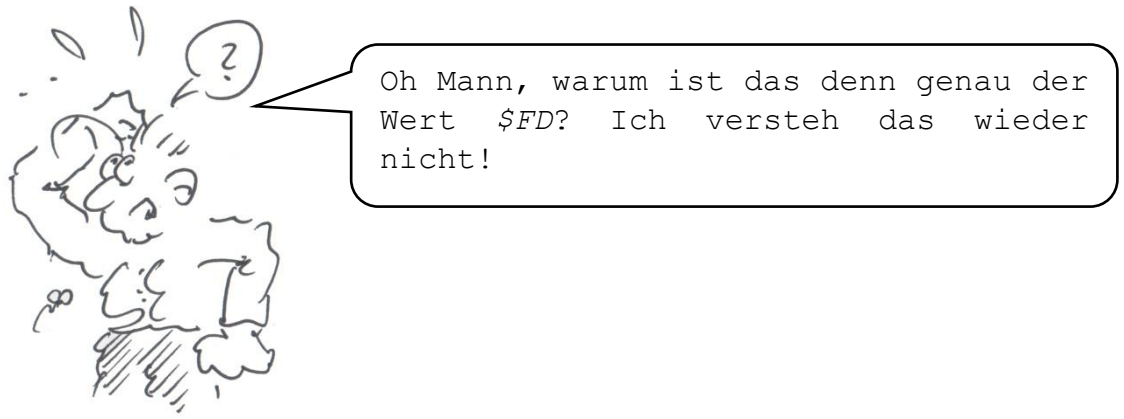
Im zweiten Teil wird das *DDRB* mit dem Wert *$00* versehen, so dass alle Pins als Eingänge arbeiten.

Abbildung 343 - Programm zur Abfrage der Taste S - Teil 2

Teil 3: Portregister A mit Wert versehen

Im dritten Teil wird das Portregister *A* (PRA) mit dem Wert *$FD* versehen, so dass hier eine Abfrage der Spalte *1* (Column *1*) erfolgt.

Abbildung 344 Programm zur Abfrage der Taste S - Teil 3



Das ist recht einfach zu verstehen, wenn du einen Blick auf die nachfolgende Abbildung wirfst.

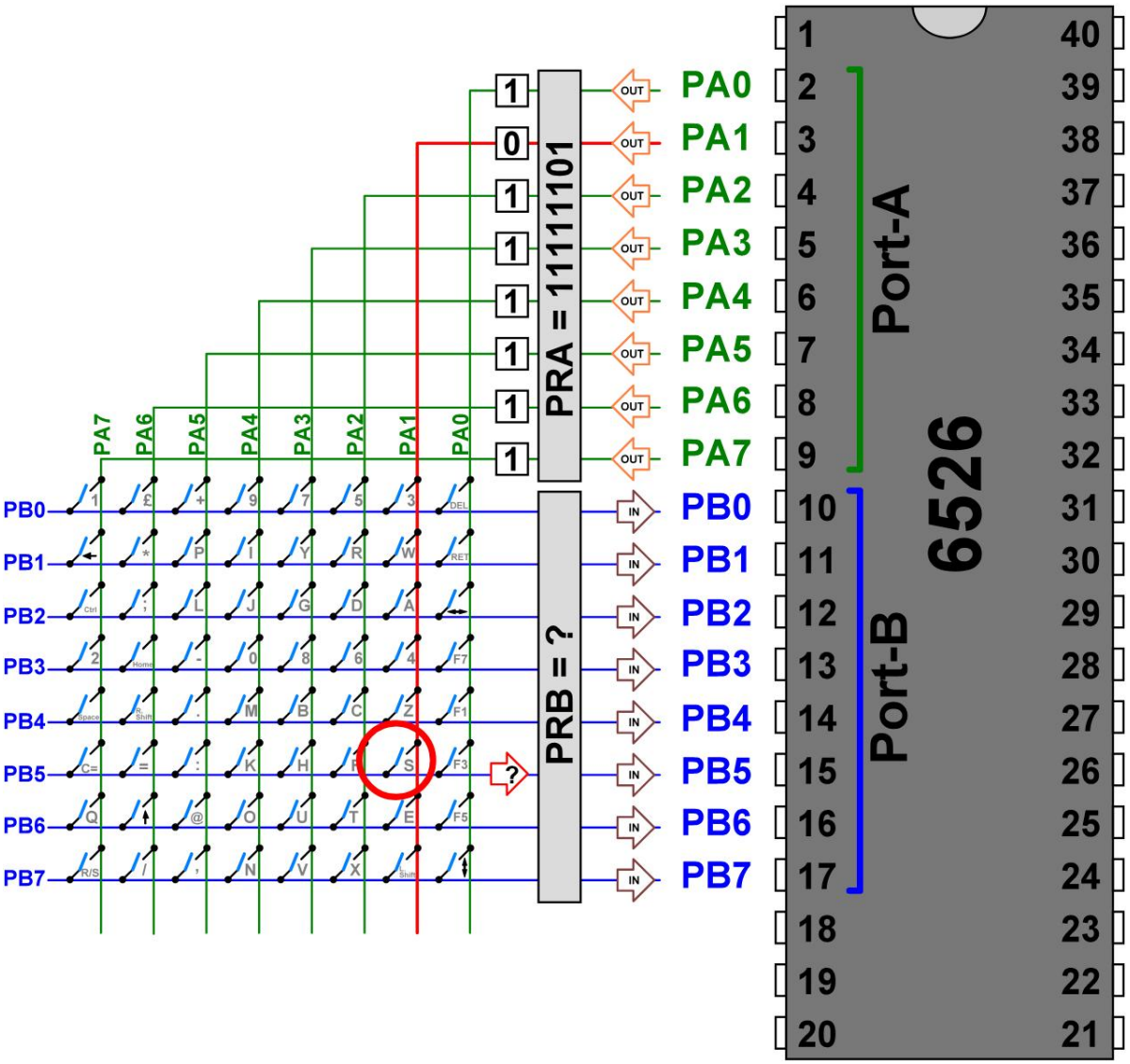


Abbildung 345 - Es wird Spalte 1 für den Test der Matrix aktiviert

Ich habe die Taste ***S*** in der Abbildung rot umrandet und wir können sehen, dass es sich um die Spalte *1* (Port A ist für die Spalten zuständig) handelt, mit der diese Taste verbunden ist. Natürlich hängen auch noch andere Tasten an dieser Spalte, wie zum Beispiel *Z* bzw. *4* darüber und *E* darunter, um nur einige zu nennen. An dieser Stelle muss ich wieder die Logikumkehr erwähnen, denn mit einer *0* wird diese Spalte aktiviert. Jetzt geht es darum, die richtige Zeile anzusprechen, an der die Taste ***S*** quasi hängt. Das sehen wir dann im nächsten Schritt. Aus der Abbildung ist ersichtlich, dass es sich um den Anschluss *PB5* von Port *B* handelt. Diesen müssen wir jetzt abfragen und das natürlich am besten innerhalb einer Schleife.

Teil 4: Portregister B kontinuierlich abfragen

Im vierten Teil wird das Portregister *B* (PRB) kontinuierlich abgefragt, um dann den gelesenen Wert auszuwerten.

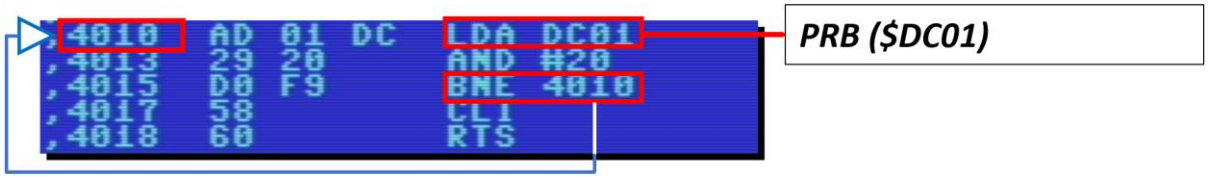


Abbildung 346 Programm zur Abfrage der Taste S - Teil 4

Wir sehen, dass es zu Beginn zu einer Abfrage des Inhaltes von Portregister *B* (*$DC01*) kommt und dieser Wert im Akku geladen wird. Jetzt geht es daran, die genannte Zeile *5*, also den Pin *PB5* auszuwerten. Erst, wenn dieser den Wert *1* vorweist, wurde die Taste ***S*** gedrückt. Die Bitkombination bzw. die Abfragemaske lautet demnach *00**1**00000*. Das erfolgt über den *AND*-Befehl mit dem Wert *#$20*. Hier noch mal zur Verdeutlichung dieses Wertes die hexadezimale Darstellung.

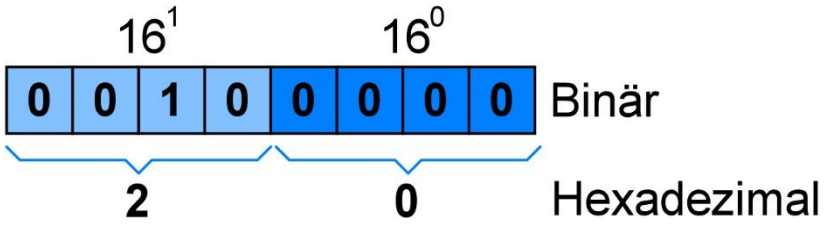


Abbildung 347 - Der Wert 00100000 in hexadezimaler Schreibweise ($20)

Nun habe ich mit dem Abfragen eines einzelnen Bits bis hier hin gewartet und das Thema nicht unter das Kapitel *Bit-Spielereien* gesetzt. Ich sagte ja, dass ich Themen erst dann zur Sprache bringe, wenn sie relevant werden.

Sehen wir uns also den *AND*-Befehl genauer an.

Nachfolgend die Matrix für den *AND*-Befehl.

OP-Code: 29

MSD	W65C02S OpCode Matrix																MSD
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	BRK s	ORA (zp,x)			TSB zp •	ORA zp	ASL zp	RMB0 zp •	PHP s	ORA #	ASL A		TSB a •	ORA a	ASL a	BBR0 r •	0
1	BPL r	ORA (zp),y	ORA (zp) *		TRB zp •	ORA zp,x	ASL zp,x	RMB1 zp •	CLC i	ORA a,y	INC A *		TRB a •	ORA a,x	ASL a,x	BBR1 r •	1
2	JSR a	AND (zp,x)			BIT zp	AND zp	ROL zp	RMB2 zp •	PLP s	AND #	ROL A		BIT a	AND a	ROL a	BBR2 r •	2

Abbildung 348 - Die OP-Code-Matrixtabelle für den Op-Code AND

Das **#** unterhalb des Befehls deutet daraufhin, dass es sich um eine unmittelbare Adressierung handelt.

N	V	-	B	D	I	Z	C
✓	-	-	-	-	-	✓	-

Tabelle 49 - Die Flag-Beeinflussung des AND-Befehls

Operation: A ∧ M → A (Der AND-Befehl überträgt den Speicher/Wert und den Akkumulator an den Addierer, der eine binäre UND-Verknüpfung durchführt und das Ergebnis im Akkumulator speichert.)

Über den *BNE*-Befehl erfolgt jetzt die Auswertung des Z-Flags. Ist der Wert nicht gleich dem angegebenen *$20*, erfolgt ein Sprung zur Adresse *$4010*. Ansonsten wird das Programm beendet, was von BASIC heraus über die Eingabe von

SYS 16384

gestartet wird. Es sei noch erwähnt, dass über den *CLI*-Befehl kurz vor der Beendigung des Programms die Interrupt-Anforderungen wieder ermöglicht werden. Nach der Eingabe des genannten Befehls verschwindet der blinkende Cursor und es wird auf den richtigen Tastendruck gewartet.

Abbildung 349 - Das Programm wurde von BASIC aus gestartet

Nach dem Druck auf die Taste ***S*** meldet sich der BASIC-Interpreter mit einem blinkenden Cursor zurück und auch das ***S*** ist zu sehen, das sich im Tastaturpuffer befand.

Abbildung 350 - Das Programm wurde von BASIC aus gestartet beendet

Machen wir doch einmal ein paar Experimente mit unterschiedlichen Werten zum Beispiel für den Port *B*. Ich habe den Code so modifiziert, dass in der Zeile, in der der *AND*-Befehl steht, den Wert von *$20* auf *$21* geändert habe.

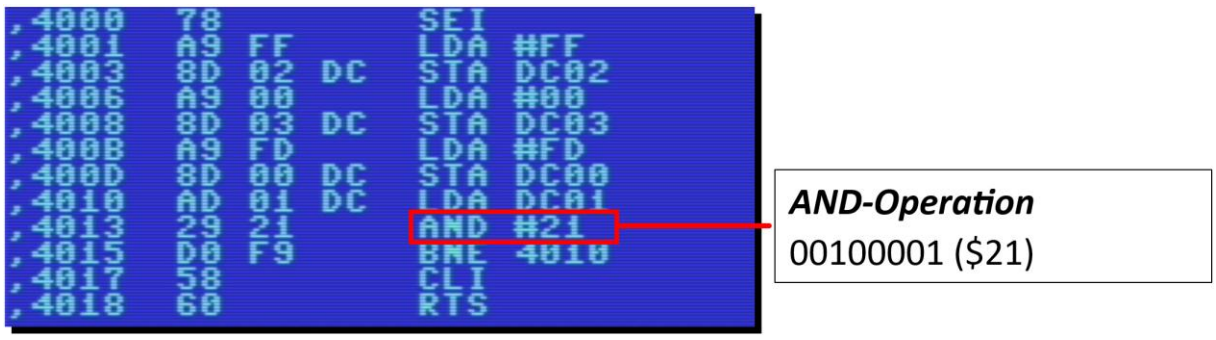


Abbildung 351 - Eine kleine Modifikation des Vergleichs

Was würde das für den Programmablauf bedeuten? Sehen wir uns das anhand der schon gezeigten Keyboard-Matrix an.

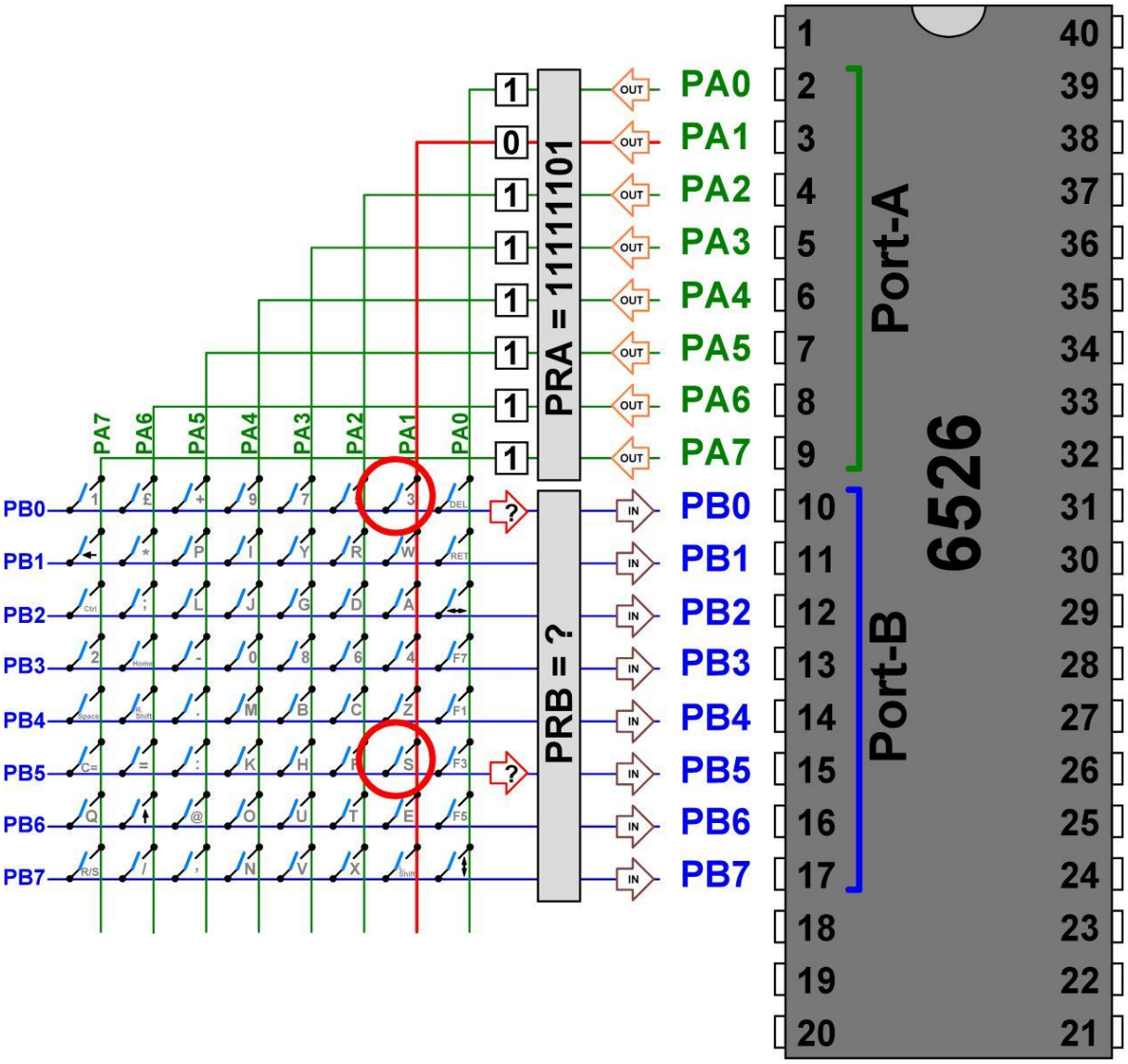


Abbildung 352 - Die modifizierte Abfrage

Es ist zwar weiterhin die Spalte *1* aktiviert, doch diesmal kommt zur Taste ***S*** noch die Taste ***3*** hinzu. Warum? Na ganz einfach, weil der Wert *$21* zusätzlich zum vorherigen Beispiel, bei dem Bit *PB5* aktiviert war, jetzt auch noch Bit *PB0* berücksichtigt. Und da es eine *AND*-Operation ist, was bedeutet das für den Programmlauf? Richtig, es müssen beide Tasten, also ***S*** und ***3*** gleichzeitig gedrückt werden, damit das Programm aus der Endlosschleife entlassen wird.

Was müssten wir anpassen, damit einer der beiden Tasten ein Verlassen der Endlosschleife bewirkt und nicht beide gleichzeitig gedrückt werden müssten? Natürlich gibt es hier wieder zahlreiche Lösungsansätze und ich habe mich für den folgenden Entschieden. Aber werfen wir zunächst einen Blick auf das Flussdiagramm

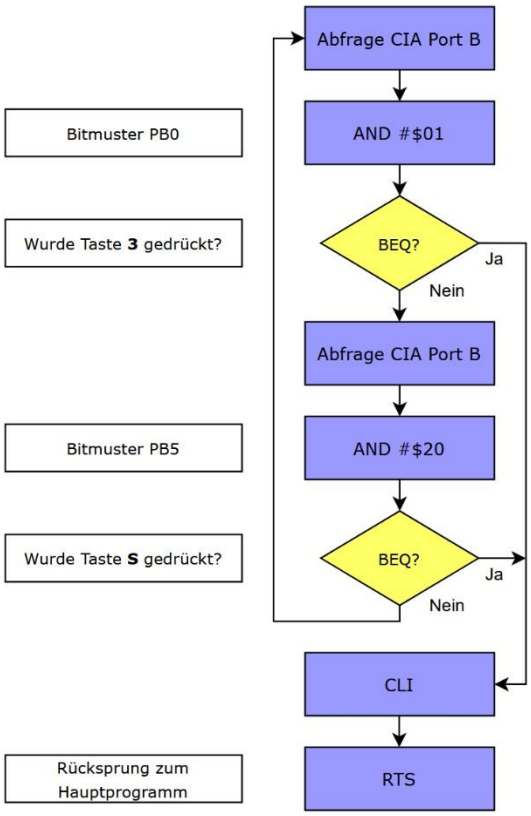


Abbildung 353 - Das Flussdiagramm für die Tastenabfragen

In einer Endlosschleife wird kontinuierlich *Port B* abgefragt. Beim ersten Mal wird über den *AND*-Befehl mit dem binären Muster *00000001* (*$01*) der Eingang *PB0* abgefragt, was für die Taste ***3*** relevant ist. Beim zweiten Mal wird über den *AND*-Befehl mit dem binären Muster *00010000* (*$20*) der Eingang *PB5* abgefragt, was für die Taste ***S*** relevant ist. Tritt keines der beiden Ereignisse ein, wird die Schleife erneut durchlaufen. Kommt es jedoch zum Drücken einer der beiden Tasten, wird diese Schleife durch den JMP-Befehl verlassen und es kommt im Endeffekt zum Verlassen des Unterprogramms und zum Rücksprung zu BASIC. Hier nun das komplette Assemblerprogramm.

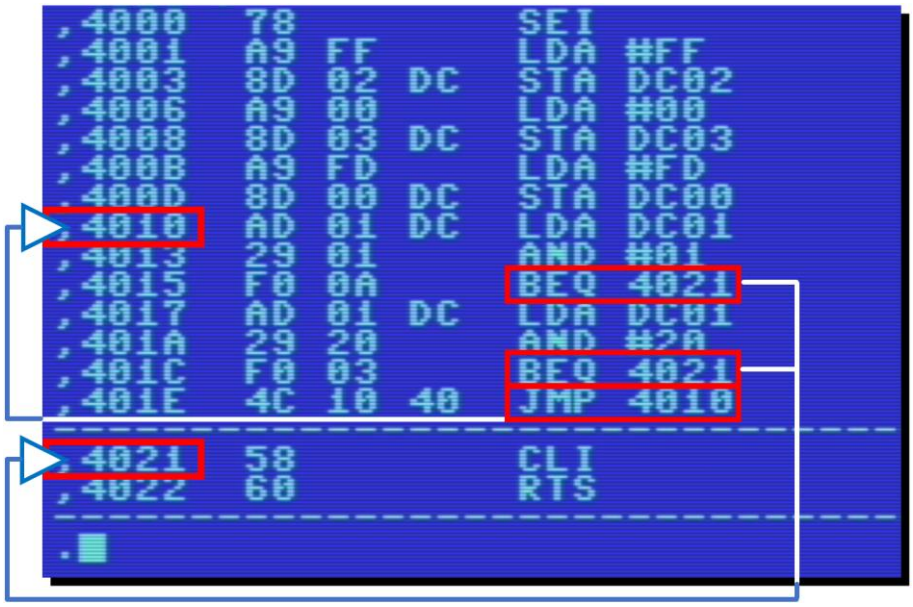


Abbildung 354 - Das Assemblerprogramm für das Abfragen der beiden Tasten

BASIC-DATA-Zeilen generieren

BASIC-Data-Zeilen generieren

Wenn man aus BASIC heraus Maschinenprogramme erstellen möchte, so ist das ebenfalls möglich. Genutzt werden kann dafür der Mechanismus, Werte hinter DATA-Zeilen abzulegen und mit dem READ-befehl zu lesen. Im Anschluss werden dann die gelesenen Werte über den *POKE*-Befehl an die richtigen Speicherstellen versendet. Das schöne bei SMON ist die Möglichkeit, aus einem bestimmten Speicherbereich heraus eine DATA-Zeile mit den korrekten Werten zu generieren. Es wird automatisch eine Zeilennummer mit der Adresse ***32000*** (definiert in den Speicherstellen *$C087/$C088*) generiert. Der allgemeine Befehl dazu lautet

Bxxxx yyyy

Die Buchstabenfolge *xxxx* steht hier für die Startadresse und *yyyy* für die Endadresse, wobei hier noch *+1* gerechnet werden muss. Sehen wir uns das an einem schon gezeigten Beispiel an. Der Assemblercode dazu lautet.

Abbildung 355 - Ein Assemblerprogramm

Wir sehen, dass das Programm den Speicherbereich von *$A4000* bis *$A00C* einnimmt. Laut Dokumentation muss jetzt zur Generierung der DATA-Zeilen der folgende Befehl eingegeben werden.

Das hat zur Folge, dass SMON verlassen wird und sich der BASIC-Interpreter mit den folgenden Zeilen zurückmeldet.

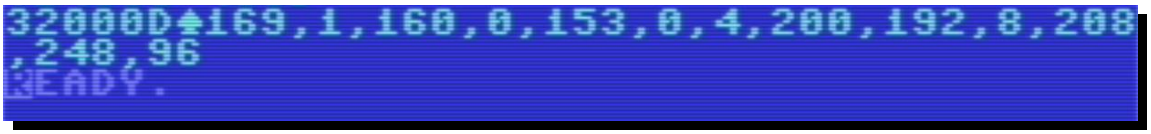


Abbildung 356 - Der BASIC-Interpreter meldet sich zurück

Auf den ersten Blick schaut das ein wenig unübersichtlich aus und auch das *Pik*-Zeichen ist sehr merkwürdig. Zudem blinkt der Cursor mitten am Anfang der *READY*-Rückmeldung. Doch keine Panik! Wir drücken einfach mit der Pfeil-Taste den Cursor eine Zeile nach unten und geben dort den *LIST*-Befehl ein.

```
LIST

32000 DATA169,1,160,0,153,0,4,200,192,8,
208,248,96
READY.
```

Abbildung 357 - Die Anzeige des Listings

Ok, das schaut schon besser aus. Was machen wir aber jetzt mit diesen Zeilen? Wenn der *RUN*-Befehl ausgeführt würde, passiert rein gar nichts. Es muss zum Lesen dieser *DATA*-Zeilen ein kleines BASIC-Programm geschrieben werden, dass die Werte über eine *FOR*-Schleife einliest und über einen *POKE*-Befehl an die richtigen Speicheradressen schreibt. Das Programm gestaltet sich in dieser Weise.

```
LIST

10 S=16384
20 FOR I=S TO S+12
30 READ D : POKE I,D
40 NEXT I
32000 DATA169,1,160,0,153,0,4,200,192,8,
208,248,96
READY.
```

Abbildung 358 - Das BASIC-Programm zum Lesen und Schreiben der Werte

Da sich unser Programm an Speicherstelle *$4000* befinden soll, muss die Startadresse in der Variablen ***S*** in Zeile *10* mit den dezimalen Wert *16384* initialisiert werden. Das Programm besitzt *13* Bytes, was bedeutet, dass die *FOR*-Schleife noch *12* weitere Bytes einlesen muss, wie das in Zeile *20* zu erkennen ist. Der *READ*-Befehl liest die Werte in der *DATA*-Zeile von links nach rechts und schreibt diese nacheinander in der Zeile *30* über den *POKE*-Befehl an die richtigen Speicheradressen. Um das korrekte Verhalten zu testen, schlage ich vor, über den Menüpunkt

File > Reset > Power cycle machine

einen komplette Aus-Einschaltvorgang des C64 zu simulieren, damit der Speicher komplette leer ist. Das muss jedoch nicht unbedingt gemacht werden, falls vorher ein anderes Assemblerprogramm im Speicher liegt.

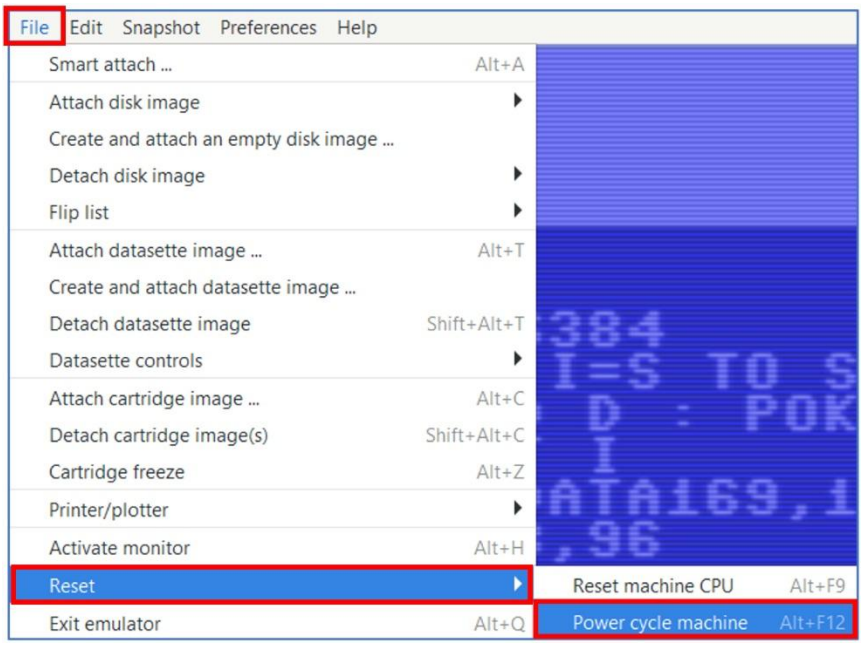


Abbildung 359 - Aus- und wieder Einschalten des C64 simulieren

Im Anschluss kann das eingegebene BASIC-Programm über den RUN-Befehl gestartet werden, was das Assemblerprogramm jedoch noch nicht startet. Das erfolgt erst über die Eingabe von

SYS 16384

Sehen wir nach, was passiert. Ich habe dazu *RUN* in der zweiten Zeile ausgeführt und alles weitere dann eingegeben.

Abbildung 360 - Das Assemblerprogramm wurde erfolgreich eingelesen und gestartet

In der ersten Zeile sind die *8* Buchstaben zu sehen, die das Assemblerprogramm im Bildschirmspeicher anzeigen sollte. Wer es hinsichtlich des BASIC-Listings natürlich übersichtlicher liebt, der kann auch die folgende Formatierung nutzen, die aber etwas mehr Aufwand bedeutet.

```
10 S=16384
20 FOR I=S TO S+12
30 READ D : POKE I,D
40 NEXT I

32000 DATA 169,1    : REM LDA #$01
32010 DATA 160,0    : REM LDY #$00
32020 DATA 153,0,4  : REM STA $0400,Y
32030 DATA 200      : REM INY
32040 DATA 192,8    : REM CPY #$08
32050 DATA 208,248  : REM BNE $4004
32060 DATA 96       : REM RTS
```

Abbildung 361 - Das aufgehübschte BASIC-Programm

Jeder Assemblerbefehl hat jetzt eine eigene *DATA*-Zeile bekommen und ist am Ende mit Kommentarzeilen mittels REM-Befehl (**REM**ark) versehen worden und gerade zu Dokumentationszwecken ist das sicherlich die bessere Variante.

Interessante Hardware

Wenn es darum geht, einen C64 auf separater Hardware zu starten, dann ist in meinen Augen das ***Turbo Chameleon 64***-Modul einfach fantastisch. Ich nenne das Modul, weil ich davon überzeugt bin und nicht, weil ich Werbung dafür machen möchte, um einem Sponsoring gerecht zu werden, was nicht stattgefunden hat.

Abbildung 362 - Das Turbo Chameleon 64-Modul - Version 1

Das Turbo Chameleon 64 - kurz TC64 - ist ein universelles Steckmodul für den Commodore C64, das in den Expansionsport gesteckt wird, aber auch als Stand-Alone, also ohne C64-Hardware, lauffähig ist. Entwickelt, hergestellt und vertrieben wird das Turbo Chameleon 64-Modul von *Individual Computers*. Nachfolgend ist die Internetadresse zum Hersteller zu finden.

	Das Turbo Chameleon 64 - Hersteller
https://icomp.de/shop-icomp/de/shop/product/Turbo Chameleon 64.html	

Einige weiterführende Informationen sind unter der folgenden Internetadresse zu finden.

	**Das Turbo Chameleon 64 - Wiki**
https://www.c64-wiki.de/wiki/Turbo_Chameleon_64	

Abschließende Worte zu Teil 1

Ich denke, dass dieser Umfang für den Teil 1 erst einmal ausreichend sein sollte. Ich würde mich über eine Rückmeldung sehr freuen und auch über ein paar anregende Kommentare oder Wünsche. In *Teil 2* - mal sehen, wann ich es schaffe - werde ich dann tiefer in die Thematik eintauchen.

Viel Spaß mit dem Büchlein

Erik Bartmann, *August 2024*