Baloui

„Letzte Chance"

# COMMODORE 64 & 128

# Maschinensprache für Einsteiger

Ein DATA BECKER Buch

Baloui

„Letzte Chance"

# COMMODORE 64 & 128

# Maschinensprache für Einsteiger

Ein DATA BECKER Buch

ISBN 3-89011-182-3

Wichtiger Hinweis:
Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

# Vorwort

Sehr geehrter Leser,

dieses Buch wendet sich an Einsteiger in die Maschinensprache-Programmierung der Commodore-Computer C64, C16, Plus/4 und C128.

Maschinensprache wird oftmals als schwierig, abstrakt usw. bezeichnet. Dieses Buch soll beweisen, daß Maschinensprache ebenso problemlos wie jede andere Programmiersprache zu erlernen ist. Theoretische Abhandlungen werden Sie relativ selten entdecken. Dieses Buch ist praxisorientiert, d.h., die verschiedenenen Befehle der Maschinensprache werden anhand ausführlich erläuterter Demoprogramme dargestellt. Sie werden niemals mit einem Befehl, der nur theoretisch behandelt wurde, allein gelassen. Immer wird Ihnen zugleich auch die praktische Anwendung des Befehls mit mehreren Beispielprogrammen vorgeführt.

Der Zweck dieses Buches besteht darin, Leser, die bereits einigermaßen mit BASIC vertraut sind, auf möglichst einfache Weise in die Maschinenprogrammierung einzuführen.

Möglichst einfach bedeutet, daß Ihnen zu Beginn nicht der Reihe nach alle Befehle dieser Sprache erläutert werden, sondern zuerst jene, die problemlos anzuwenden sind und mit denen bereits effektive Programme erstellt werden können. Anschließend werden die wichtigsten Routinen des Betriebssystems vorgestellt, die Ihnen fix und fertig programmiert viele Hilfsprogramme zur Verfügung stellen, die Sie in eigenen Programmen verwenden können.

Nach diesen Kapiteln werden Sie bereits in der Lage sein, sehr interessante Programme zu erstellen. Diesem einführenden Teil folgt der zweite Hauptteil, in dem nach einer kurzen Wiederholung des im ersten Teil erlernten Stoffes systematisch alle Befehlsklassen erläutert werden, die zur Erstellung komplexer Maschinenprogramme benötigt werden.

Dieser zweite Hauptteil ist wie folgt aufgebaut:

- Befehlserläuterung anhand von Demoprogrammen: Die Wirkungsweise und Funktion des jeweiligen Befehls wird erklärt und der Befehl selbst zur Vertiefung in mehreren kleinen Programmen angewendet.

- Vertiefung durch Übungsprogramme: Jedem in sich abgeschlossenen Kapitel folgt ein Abschnitt, in dem ausschließlich Programme erstellt werden. Diese zum Teil recht anspruchsvollen Programme sollen Ihnen anschaulich zeigen, wie die erläuterten Befehle sinnvoll einzusetzen sind und Ihnen die benötigten Programmiertechniken vermitteln.

Im Gegensatz zu höheren Programmiersprachen wie BASIC ist die Kenntnis der vorhandenen Befehle allein sinnlos. Um in Maschinensprache programmieren zu können, müssen Sie wissen, wie Ihr Rechner intern aufgebaut ist, welche wichtigen Bereiche er enthält, wie die Routinen des Betriebssystems genutzt werden und welche Speicherbereiche für die Ablage von Daten und Programmen vorhanden sind.

Im Verlaufe des Buches werden Sie feststellen, daß es bereits mit relativ geringen und schnell erlernten Kenntnissen über das Innenleben Ihres Rechners möglich ist, sehr interessante Programme zu entwickeln.

In vielen Büchern über Maschinensprache werden zwar Befehle und Adressierungsarten erläutert, jedoch nicht speziell auf den Rechner des Lesers bezogen. Der erlernte Stoff befindet sich gewissermaßen im luftleeren Raum und kann nicht umgesetzt werden.

Um diesen Fehler zu vermeiden, wird in allen (!) Kapiteln speziell auf Commodore-Rechner eingegangen und deren Programmierung in Maschinensprache. Der dritte Teil beschäftigt sich vorwiegend mit den Besonderheiten, die bei der Programmierung des C16, Plus/4 oder des C128 zu berücksichtigen sind.

Der vierte Hauptteil erläutert ein ganz hervorragendes Hilfsmittel, das uns der Rechner kostenlos zur Verfügung stellt, den eingebauten BASIC-Interpreter.

Die Erläuterung der wichtigsten Routinen des BASIC-Interpreters erschließt Ihnen zwei Welten: BASIC und Maschinensprache. Maschinenprogramme stehen nicht mehr im freien Raum, sondern können zusammen mit BASIC-Programmen eingesetzt werden, um z.B. Eingabe- oder Sortierroutinen zu erstellen.

Das vorliegende Buch bezieht sich zunächst auf den Speicheraufbau des C64. Im ersten Hauptteil dieses Buches werden jedoch Demoprogramme jeweils in mehreren Versionen für verschiedene Commodore-Heimcomputer vorgestellt. Im Text wird an Stellen, die den Speicheraufbau betreffen, beschrieben, wo sich die entsprechenden Bereiche beim C16, C116, Plus/4 und C128 befinden und welche Besonderheiten bei diesen Rechnern eventuell zu berücksichtigen sind.

Für jeden Besitzer eines dieser Computer besonders wertvoll ist der dritte Teil, der sich mit der Speicherorganisation des C16, C116, des Plus/4 und des C128 befaßt. Durch diesen Aufbau bezieht sich der im vorliegenden Buch vermittelte Lehrstoff auf alle Commodore-Heimcomputer.

Noch ein Hinweis: Am Ende dieses Buches finden Sie das komplette Listing eines Monitor-Programms, eines sehr nützlichen Werkzeugs bei der Programmierung.

Ludwigshafen, im Dezember 1986 Said Baloui

# Inhaltsverzeichnis

# Einleitung

Ohne Ihr Wissen haben Sie ziemlich sicher bereits das eine oder andere Maschinensprache-Programm eingegeben. In fast allen Heimcomputer-Zeitschriften finden sich Unmengen von Listings, die zum Teil aus sogenannten Data-Zeilen bestehen. Diese Data-Zeilen enthalten fast immer Maschinensprache-Programme (im folgenden kurz Maschinenprogramme genannt).

Diese Programme demonstrieren häufig den großen Vorteil von Maschinensprache, die enorme Geschwindigkeit. Die erstellten Programme sind meistens mehrere hundertmal so schnell wie BASIC-Programme und somit immer noch extrem viel schneller als selbst compilierte BASIC-Programme.

Ein weiterer Vorteil: Ein Maschinenprogramm ist immer erheblich kürzer als das entsprechende BASIC-Programm. Mit keiner anderen Programmiersprache kann ein derart kompakter und effizienter Programmcode erstellt werden.

Einige Probleme - z.B. sogenannte Interrupt-Routinen - lassen sich sogar nur in Maschinensprache lösen. Sie sehen also, es gibt genug Gründe, um sich für diese Programmiersprache zu interessieren. Daß es nur in Maschinensprache möglich ist, auf Heimcomputern wirklich professionelle Programme zu schreiben, beweist die Werbung für viele Programme, in denen diese als "zu 100% in Maschinensprache geschrieben" gerühmt werden.

Ein kleiner Nachteil ist, daß Maschinensprache-Befehle nicht ganz so komfortabel sind wie BASIC-Befehle. Zum Beispiel müssen zur Division oder Multiplikation von zwei Zahlen eigene Unterprogramme erstellt werden, entsprechende Befehle sind nicht vorhanden. Dieser Nachteil ist jedoch nicht sehr gravierend, da Sie wohl kaum eine Buchhaltung in Maschinensprache schreiben, sondern eher Videospiele oder eine kleine Textverarbeitung. Gerade zeitkritische Anwendungen dieser Art sind optimale Einsatzgebiete für Maschinensprache.

Zudem werden wir im Verlaufe dieses Buches die sogenannten Betriebssystem- und BASIC-Interpreter-Routinen kennenlernen, fest eingebaute Unterprogramme, die das Erstellen eigener Programme erheblich erleichtern. Diese Routinen stellen eine Art Unterprogramm-Bibliothek dar und enthalten fertige Programmteile für die verschiedensten Anwendungen und Problemstellungen. Die Nutzung dieser Unterprogramme versetzt Sie sehr schnell in die Lage, effektive Maschinenprogramme zu schreiben.

Sicherlich kennen Sie Hilfsmittel zur Programmierung in BASIC, wie z.B. Befehlserweiterungen. Zur Programmierung in Maschinensprache benötigen Sie einen sogenannten Monitor, ein Programm, das die Befehlseingabe ungemein erleichtert. Monitor-Programme gibt es in Hülle und Fülle, z.B. als Listings in den verschiedensten Zeitschriften. Die Arbeitsweise und Bedienung dieser Programme ist standardisiert, die benötigten Eingaben unterscheiden sich nur in seltenen Fällen. Sowohl im C16 und Plus/4 als auch im C128 ist ein solches Programm fest eingebaut.

Ein Monitor ist Voraussetzung für die Eingabe der in diesem Buch verwendeten Programmbeispiele. Im Anhang dieses Buches finden Sie ein Monitor-Programm, das alle C64- und C128-Besitzer (im 64er-Modus) verwenden können.

# Teil 1: Einführung in Assembler

Wie bereits im Vorwort erläutert, bildet der erste Teil dieses Buches eine Art Grundkurs.

Der Grundkurs vermittelt Ihnen bereits alles Notwendige zur Erstellung kleiner Assembler-Programme, die Grundlagen zum Umgang mit einem Monitor-Programm, die wichtigsten Befehle und Adressierungsarten und erläutert die wichtigsten Unterprogramme, die uns das Betriebssystem zur Verfügung stellt.

## 1. Die ersten Gehversuche

Für den Fall, daß Sie noch nie (auch nicht in Form von Data-Zeilen) Kontakt mit einem Maschinenprogramm hatten, sollten Sie das weiter unten abgebildete Demoprogramm eingeben.

Das eigentliche Maschinenprogramm ist in der Data-Zeile am Ende des BASIC-Programms enthalten. Seine recht einfache Aufgabe besteht darin, die oberen Bildschirmzeilen mit dem Buchstaben A zu füllen.

Besitzer eines C64 können dieses Programm sofort eingeben und wie gewohnt mit RUN starten. Wenn Sie mit einem C16, C116 oder einem Plus/4 arbeiten, geben Sie bitte die zweite Programmversion ein.

Version 1 (C64, C128 im 64er-Modus)

```
100 REM *** 'A' AUSGEBEN ***
110 FOR I=49152 TO 49152+10
120 : READ A:POKE I,A
130 NEXT
140 PRINT CHR$(147):REM SCREEN LOESCHEN
150 SYS 49152:REM PROGRAMM STARTEN
160 END
170 :
180 DATA 169,1,162,0,157,0,4,202,208,250,96
```

Version 2 (C16, C116, Plus/4)

```
100 REM *** 'A' AUSGEBEN ***
110 FOR I=1015 TO 1015+10
120 : READ A:POKE I,A
130 NEXT
140 PRINT CHR$(147):REM SCREEN LOESCHEN
150 SYS 1015:REM PROGRAMM STARTEN
160 END
170 :
180 DATA 169,1,162,0,157,0,12,202,208,250,96
```

Ein Fall für sich ist der C128. Dieser Rechner ist zweifellos von allen Commodore-Heimcomputern der leistungsfähigste (wenn man den Amiga nicht ebenfalls als Heimcomputer bezeichnet). Dafür ist jedoch die Programmierung in Maschinensprache aufwendiger.

Vor allem die Ausgabe von Zeichen auf dem Bildschirm ist beim C128 in Maschinensprache problematisch. Wenn Sie einen C128 besitzen, sollten Sie daher den 64er-Modus zum Bearbeiten dieses Buches verwenden. Als Entschädigung für diese Unbequemlichkeit wird Ihnen ein eigenes Kapitel zeigen, welche Besonderheiten dieser Rechner bietet und wie Sie mit diesen praktisch umgehen (Bankswitching, nicht direkt adressierbarer Bildschirmspeicher).

Zurück zu unserem Demoprogramm. BASIC-Programmierer (und an die wendet sich ja dieses Buch) sind sicherlich über die enorme Geschwindigkeit erstaunt, mit der die 256 Zeichen ausgegeben werden. Irgendeine Verzögerung ist nicht festzustellen, das Programm arbeitet für das menschliche Auge praktisch in Nullzeit.

Zum Vergleich ein BASIC-Programm, das analog dem Maschinenprogramm arbeitet:

```
100 REM *** 'A' AUSGEBEN IN BASIC ***
110 PRINT CHR$(147);:REM SCREEN LOESCHEN
120 FOR I=1 TO 256
130 : PRINT"A";
140 NEXT
```

Zugegeben: auch das BASIC-Programm ist nicht allzu langsam (circa eine Sekunde zur Ausgabe aller A's). Der Unterschied zwischen diesem und dem Maschinenprogramm ist dennoch erstaunlich.

Um Ihnen den Geschwindigkeitsunterschied zu verdeutlichen: eine vor Jahren von mir in BASIC geschriebene Sortierroutine (Bubble-Sort) benötigte zur Sortierung von 1000 Strings etwa drei Stunden. Die gleiche Routine in Maschinensprache sortiert 1000 Strings in zehn Sekunden!

## 2. Maschinensprache und Assembler

Der Begriff Maschinensprache fiel bereits ziemlich oft, ohne daß eine nähere Erläuterung erfolgte. Um dieses Versäumnis nachzuholen, benötigen wir ein wenig trockene Theorie.

Die Maschinensprache ist die einzige (!) Sprache, die unser Computer wirklich versteht. Wie wir noch sehen werden, versteht er BASIC nicht, BASIC ist eine Fremdsprache für ihn. Die eigentliche Maschinensprache besteht nicht aus Befehlswörtern, sondern aus Zuständen von Leitungen. Entweder fließt Strom oder es fließt kein Strom. Diese beiden Zustände einer Leitung

kann unser Rechner unterscheiden (verstehen) und darauf reagieren.

Der eigentliche Rechner besteht aus der sogenannten CPU (Central Processing Unit, Zentraleinheit). Diese Zentraleinheit versteht Befehle, die sich aus der Kombination der Zustände mehrerer Leitungen ergeben.

Diese Erläuterung klingt kompliziert, ist es jedoch nicht. Stellen wir uns acht Leitungen vor, die mit dem Gehirn unseres Computers - der CPU - verbunden sind.

```
Leitung 1 ----------------------------------------
Leitung 2 ----------------------------------------
Leitung 3 ----------------------------------------
Leitung 4 ----------------------------------------
Leitung 5 ----------------------------------------
Leitung 6 ----------------------------------------
Leitung 7 ----------------------------------------
Leitung 8 ----------------------------------------
```

Jede dieser acht Leitungen besitzt zwei mögliche Zustände: Strom fließt/kein Strom fließt. Diese beiden Zustände kennzeichnen wir mit je einer Zahl, eins (Strom fließt) und null (kein Strom fließt).

Insgesamt ergeben sich 256 mögliche Kombinationen von Nullen und Einsen, z.B.

```
        00000000
oder    00000001
oder    00000010
oder    10010010
oder    11101110
oder    11111111
...
...
```

Jede dieser Kombinationen wird von der CPU als ein Befehl interpretiert. Wie ein von uns erteilter Befehl die Kombination dieser Leitungszustände herstellt, interessiert uns nicht weiter.

Wichtig für uns ist jedoch, daß diese maximal 256 Befehle ähnlich wie BASIC-Befehle Namen besitzen, die wir bei der Programmierung eingeben.

Ein Beispiel für einen solchen Befehl ist der Befehl LDA, eine Abkürzung für Load Accumulator. Diese Befehle werden Assembler-Befehle genannt und mit ihnen werden wir im folgenden programmieren. Anstelle des Ausdrucks Maschinensprache-Programm sollten wir daher besser von Assembler-Programm sprechen, da sich das Programm aus einzelnen von uns eingegebenen Assembler-Befehlen zusammensetzt.

Für Sie ist nun sicher interessant, inwieweit sich die Assembler-Befehle des C16, C116, des Plus/4, des C64 und des C128 voneinander unterscheiden. Die Antwort lautet: überhaupt nicht!

Welche Assembler-Befehle ein Computer besitzt, hängt von der eingebauten CPU ab. Beispielsweise besitzt der C64 eine CPU mit dem Namen 6510, der C128 die CPU 8502. Glücklicherweise stammen die CPU's aller genannten Commodore-Rechner vom gleichen Hersteller und sind kompatibel, d.h., sie verstehen die gleichen Befehle.

Dank dieser Kompatibilität gelten alle Befehlsbeschreibungen dieses Buches für jeden Commodore-Heimcomputer, im Gegensatz zur üblicheren Programmiersprache BASIC, von der bekanntlich der C64 eine andere Version besitzt als der C16 und Plus/4, und deren BASIC-Version unterscheidet sich wiederum von jener des C128.

Die Frage ist nun, wie Assembler-Programme eingegeben werden. Mit dem eingebauten BASIC-Interpreter ist die Eingabe nicht möglich, wie Sie sofort feststellen, wenn Sie z.B. den Assembler-Befehl LDA eingeben und RETURN drücken.

Zur Eingabe verwenden wir das erwähnte Monitor-Programm. Einen solchen Monitor müssen Sie nicht unbedingt kaufen. C16, C116, Plus/4 und C128 besitzen einen eingebauten Monitor, den C128-Besitzer jedoch nicht verwenden können, da sie dieses Buch ja im 64er-Modus bearbeiten sollen.

Der C64 besitzt zwar keinen Monitor, für diesen Rechner existiert jedoch eine Unmenge an Listings derartiger Programme in den verschiedensten Fachzeitschriften. Außerdem finden Sie im Anhang dieses Buches das Listing eines Monitors, der für den C64 und den C128 im 64er-Modus gedacht ist.

Bitte beachten Sie bei der Benutzung eines Monitors für den C64 folgendes:

1. Viele Beispielprogramme sind im Bereich $C000 geschrieben. Genau in diesem Bereich liegen aber viele Monitor-Programme. Sie benötigen also entweder einen Monitor, der von vornherein in einem anderen Bereich liegt, oder müssen den Monitor von $C000 in einen anderen Bereich verlegen, (was nicht ganz einfach ist) oder Sie schreiben die Beispielprogramme in einem anderen Bereich, beispielsweise $6000. Den hinten abgedruckten Maschinensprache-Monitor können Sie problemlos benutzen, weil er im Bereich ab $9000 liegt.

2. Der hinten abgedruckte Monitor unterscheidet sich etwas von beispielsweise dem vorhandenen Monitor des C128. Um beim Monitor des C128 eine Zeile einzugeben, müssen Sie ein A vor der gewünschten Speicherstelle eingeben und nachdem Sie RETURN gedrückt haben, erscheinen zusätzlich die Zahlen, die der Monitor für die Befehle in die Speicherstellen schreibt.

Der hinten abgedruckte Monitor gibt diese Zahlen nicht aus und versteht das A für Assemble nicht. Bei diesem Monitor geben Sie neue Befehle ein, indem Sie die gewünschte Speicherstelle entweder mit D Disassemblieren und den gewünschten Befehl an die Stelle des vorhandenen schreiben. Sie können auch einfach ein Komma vor die gewünschte Speicherstelle und dahinter den

Befehl schreiben. Wir wollen diesen Unterschied kurz an einem Beispiel zeigen:

Eingabe des Assemblerbefehls LDA $0300 beim C128-Monitor:

```
A C000 LDA $0300
```

Nach dem RETURN wird die Zeile umgewandelt und der Bildschirm zeigt:

```
A 0C000 AD 00 03 LDA $0300
A 0c003
```

Beim abgedruckten und vielen anderen Monitoren des C64 geben Sie ein:

```
,C000 LDA $0300
```

Nach RETURN zeigt der Bildschirm:

```
,C000 LDA $0300
,C003
```

(Vor beiden Zeilen steht noch ein Punkt, den der Monitor automatisch ausgibt.)

Wenn Sie also den abgedruckten oder einen ähnlichen Monitor verwenden, so ändern Sie die Eingaben bitte entsprechend um.

Wenn Sie nun ein solches Programm eingetippt und gestartet oder aber - C16, C116 und Plus/4 - den Befehl MONITOR eingegeben haben, erscheint - mit leichten Variationen je nach Monitor-Programm - folgende Einschaltmeldung:

```
   PC  IRQ  SR AC XR YR SP
;E5CF EA31 22 00 00 0A F2
```

Um diese Einschaltmeldung kümmern wir uns vorläufig nicht weiter. Am Anfang der folgenden Zeile befindet sich der Cursor. Das Monitor-Programm wartet nun auf einen Befehl. Sollten

Sie einen C64 besitzen, geben Sie bitte folgenden Befehl ein (und bestätigen Sie ihn wie üblich mit RETURN):

```
A C000 LDA #$01
```

Die eingegebene Zeile wird sofort in ein anderes Format umgewandelt:

```
A C000 A9 01    LDA #$01
A C002
```

Der Cursor befindet sich nun hinter der Zahl C002 und das Programm wartet auf eine weitere Eingabe. Geben Sie bitte der Reihe nach ebenso wie LDA #$01 die folgenden Befehle ein:

```
LDX #$00
STA $0400,X
DEX
BNE $C004
RTS
```

Nach jeder Eingabe wird in der nächsten Zeile eine Zahl vorgegeben und auf den nächsten Befehl gewartet. Betrachten Sie diese Vorgabe als eine Art AUTO-Befehl, der Ihnen nach Eingabe der ersten Zeilennummer eines BASIC-Programms alle weiteren Zeilennummern automatisch vorgibt.

Haben sie den letzten Befehl RTS eingegeben, drücken Sie bitte RETURN, ohne einen weiteren Befehl einzugeben, um die Zeilenvorgabe abzubrechen. Der Bildschirm zeigt nun folgendes Programm:

```
A C000 A9 01    LDA #$01
A C002 A2 00    LDX #$00
A C004 9D 00 04 STA $0400,X
A C007 CA       DEX
A C008 D0 FA    BNE $C004
A C00A 60       RTS
A C00B
```

Sollten Sie einen Fehler begangen haben, gehen Sie bitte mit dem Cursor in die betreffende Zeile und korrigieren Sie den Assembler-Befehl wie in einem BASIC-Programm. Wichtig: korrigieren Sie bitte weder die Zeilennummer noch die folgenden zweistelligen Zahlen, sondern immer nur den von Ihnen eingegebenen Assembler-Befehl!

Geben Sie anschließend den Buchstaben X ein und drücken Sie RETURN. Die Eingabe X führt zum Verlassen des Monitors und der BASIC-Interpreter meldet sich wie üblich mit READY.

Besitzer eines C16, C116 oder Plus/4 wandeln die Eingaben wie folgt ab:

Erster Befehl:

```
A 03F7 LDA #$01
```

Folgende Befehle:

```
LDX #$00
STA $0C00,X
DEX
BNE $03FB
RTS
```

Sie sollten folgendes Bild sehen:

```
A 03F7 A9 01    LDA #$01
A 03F9 A2 00    LDX #$00
A 03FB 9D 00 0C STA $0C00,X
A 03FE CA       DEX
A 03FF D0 FA    BNE $03FB
A 0401 60       RTS
```

Sie haben nun ohne den Umweg über die Data-Zeilen mit dem Monitor jenes Demoprogramm eingegeben, das 256mal das Zeichen A auf dem Bildschirm ausgab und damit Ihr erstes Assembler-Programm geschrieben. Starten Sie dieses Programm

bitte mit dem gleichen Aufruf, den auch das BASIC-Programm verwendete.

```
C64: SYS 49152
C16, C116, Plus/4: SYS 1015
```

Sie können dieses Programm übrigens ebenso wie jedes BASIC-Programm mit diesem SYS-Befehl beliebig oft starten.

## 3. Dualsystem und Hexadezimalsystem

Vielleicht wundern Sie sich darüber, daß ich Zeichenfolgen wie C000 oder 03F7 als Zahlen bezeichne. Es sind tatsächlich Zahlen, jedoch keine Dezimal- sondern sogenannte Hexadezimalzahlen. Unglücklicherweise können wir bei der Assembler-Programmierung mit unserem gewohnten Dezimalsystem nur wenig anfangen. Üblich ist die Verwendung des dualen und - wie bereits der Monitor zeigt - vor allem des hexadezimalen Systems.

Im folgenden werde ich zuerst direkt auf das duale und das hexadezimale Zahlensystem eingehen. Anhand konkreter Beispiele werden die Unterschiede zum Dezimalsystem erläutert.

Das Dualsystem ist die Grundlage jeglicher "Computerei". Für jede Stelle einer Dualzahl steht nur eine von zwei verschiedenen Ziffern zur Verfügung. Diese Ziffern sind 0 und 1 (zum Vergleich: im Dezimalsystem stehen bekanntlich zehn Ziffern zur Verfügung, 0,1,...,8,9).

Warum das Dualsystem sehr gut zur Zahlenverarbeitung durch einen Computer geeignet ist, wird deutlich, wenn wir uns erinnern, daß die CPU nur zwei verschiedene Leitungszustände unterscheiden kann, "Strom fließt" und "Strom fließt nicht", die durch eben jene beiden Ziffern 0 und 1 beschrieben werden können.

Eine Stelle einer Dualzahl, die wie erwähnt aus der Ziffer 0 oder der Ziffer 1 bestehen kann, nennt man ein Bit. Acht solcher Bits bilden ein Byte. Die Bits einer Dualzahl sind von

rechts nach links durchnumeriert, wobei das erste Bit (ganz rechts) die Nummer null erhält (Bit 0, Bit 1,...,Bit 7).

| Bit 7 | Bit 6 | Bit 5 | Bit 4 | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

Die Tabelle enthält vier Bytes, d.h., vier aus je acht Bits bestehende Dualzahlen. Die Frage ist nun, welchen Wert diese Dualzahlen besitzen. Der Wert hängt offensichtlich von der Kombination aus Nullen und Einsen ab. Enthält ein Bit die Ziffer 0, nennt man es gelöscht. Ein gesetztes Bit wird durch die Ziffer 1 gekennzeichnet.

Um den Wert einer Dualzahl zu ermitteln, genügt es offensichtlich nicht, alle gesetzten Bits zu addieren. Ebenso wie im Dezimalsystem ist der Wert einer Ziffer davon abhängig, an welcher Stelle sich die Ziffer in der Zahl befindet.

Um beim Dezimalsystem zu bleiben: mit jeder Stelle weiter links verzehnfacht sich der Wert einer Ziffer. Die Ziffer 5 besitzt an der ersten Stelle den Wert fünf, an der zweiten Stelle den Wert 50, an der dritten Stelle den Wert fünfhundert und so weiter.

Im Unterschied zum Dezimalsystem verdoppelt (!) sich der Wert einer Ziffer im Dualsystem, wenn die Ziffer (0 oder 1) um eine Stelle nach links verschoben wird. Die erste Dualzahl der Tabelle (00000000) besitzt natürlich den Wert null. Die zweite Dualzahl (00000001) besitzt den Wert eins, die dritte Dualzahl (00000010) den Wert zwei und die vierte Dualzahl (00000100) den Wert vier.

Wenn man dieses Schema der Verdoppelung fortsetzt, ist leicht zu erkennen, daß bei nur einem gesetzten Bit der größte mit einer achtstelligen Dualzahl darstellbare Wert die Zahl 128 ist (10000000).

Sind mehrere Bits eines Bytes gesetzt, ergibt sich der Gesamtwert der Zahl, indem - ebenso wie im Dezimalsystem - die Werte aller Ziffern - abhängig von der Stelle in der Zahl - addiert werden.

Die Dezimalzahl 123 besitzt den Wert einhundertdreiundzwanzig, da die Ziffer 3 an der ersten Stelle den Wert drei besitzt, die Ziffer 2 an der zweiten Stelle den Wert zwanzig und die Ziffer 1 an der dritten Stelle den Wert einhundert. Addiert man diese Zahlen, ergibt sich der Gesamtwert.

Das Problem an dieser Analogie ist natürlich, daß wir aufgrund unseres ständigen Umgangs mit dem Dezimalsystem diese Berechnungen nicht mehr bewußt vornehmen, sondern die Ziffernfolge 1, 2, 3 (123) bereits als die Zahl einhundertdreiundzwanzig zu sehen glauben (was nicht stimmt, sondern nur bedeutet, daß die erforderlichen Berechnungen unbewußt ablaufen).

Versuchen Sie bitte, sich diese Berechnung des Wertes einer Zahl bewußt zu machen, indem Sie versuchen, eine Ziffer nicht sogleich als Zahl zu betrachten, sondern als Zeichen (!), dessen zahlenmäßiger Wert von seiner Position abhängt.

Gelingt Ihnen dies, ist die Übertragung dieses Schemas auf Dualzahlen ein Kinderspiel. Beispielsweise besitzt die Dualzahl 00000011 den dezimalen Wert drei, da die Ziffer 1 an der ersten Stelle den Wert eins besitzt und an der zweiten Stelle den Wert zwei.

Sehr einfach wird die Berechnung des Wertes einer Dualzahl mit Hilfe einer Tabelle, die den Stellenwert eines gesetzten Bits an den verschiedenen Stellen angibt.

```
Stellenwerte im Dualsystem

Bit 0 gesetzt => Wert   1
Bit 1 gesetzt => Wert   2
Bit 2 gesetzt => Wert   4
Bit 3 gesetzt => Wert   8
Bit 4 gesetzt => Wert  16
Bit 5 gesetzt => Wert  32
Bit 6 gesetzt => Wert  64
Bit 7 gesetzt => Wert 128
```

Zur Berechnung schauen Sie sich an, welche Bits gesetzt sind und addieren die betreffenden Stellenwerte.

Die Dualzahl 00010011 entspricht somit der Dezimalzahl 19 (1+2+16) und die Dualzahl 11111111 der Dezimalzahl 255 (1+2+4+8+16+32+64+128). Sie sehen, mit einem Byte - also einer achtstelligen Dualzahl - lassen sich durch Kombinationen gesetzter und gelöschter Bits alle Zahlen zwischen null (alle Bits gelöscht: 00000000) und 255 darstellen (alle Bits gesetzt: 11111111).

Das Hexadezimalsystem ist gewöhnungsbedürftiger als das Dualsystem, da es äußerst ungewohnte Ziffern enthält. Nicht nur 0 und 1 wie das Dualsystem oder 0,1..,8,9 wie das Dezimalsystem, sondern die Ziffern 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

Die Ziffern A..F besitzen die Ziffernwerte zehn bis 15 (A=10, B=11, C=12, D=13, E=14, F=15). Das Hexadezimalsystem besitzt einen wesentlichen Vorteil gegenüber dem Dual- und dem Dezimalsystem. Auch sehr große Zahlen können mit wenigen Ziffern dargestellt werden. Üblicherweise werden wir mit vierstelligen Hexadezimalzahlen arbeiten (Beispiel: C000, A000, 1AB5).

Im Hexadezimalsystem versechzehnfacht (!) sich der Wert einer Ziffer mit jeder Stelle weiter links. Das heißt, die Ziffer 1 besitzt an der ersten Stelle (0001) den Wert eins, an der zweiten Stelle (0010) den Wert 16, an der dritten Stelle (0100) den Wert 256 und an der vierten Stelle (1000) bereits den Wert 4096.

Der Wert einer Hexadezimalzahl ergibt sich wiederum aus der Addition der Werte der einzelnen Ziffern unter Berücksichtigung ihrer Stellung in der Zahl. Um Ihnen die Berechnung zu erleichtern, folgt eine Tabelle der Stellenwerte des Hexadezimalsystems.

| Stelle | Stellenwert |
|---|---|
| 1 | 1 |
| 2 | 16 |
| 3 | 256 |
| 4 | 4096 |

Bei der Berechnung des Wertes einer Hexadezimalzahl gehen Sie so vor:

1. Multiplizieren Sie den Ziffernwert mit dem jeweiligen Stellenwert, um den Wert der Ziffer an der betreffenden Stelle zu ermitteln.

2. Addieren Sie alle ermittelten Werte.

Beispiele (Die Berechnungen erfolgen immer von rechts nach links!):

1. 0011 => 1*1 + 1*16 + 0*256 + 0*4096 = 17
2. 1100 => 0*1 + 0*16 + 1*256 + 1*4096 = 4112
3. 1111 => 1*1 + 1*16 + 1*256 + 1*4096 = 4369
4. 00FF => 15*1 + 15*16 + 0*256 + 0*4096 = 255
5. FF00 => 0*1 + 0*16 + 15*256 + 15*4096 = 65280
6. FFFF => 15*1 + 15*16 + 15*256 + 15*4096 = 68880
7. ABCD => 13*1 + 12*16 + 11*256 + 10*4096 = 43981

Sie sehen, mit einer vierstelligen Hexadezimalzahl lassen sich Zahlen darstellen, zu deren Darstellung im Dezimalsystem bereits sechs Stellen benötigt werden.

Beim Umgang mit verschiedenen Zahlensystem stellt sich das Problem der Unterscheidung. Wie erkennen wir, ob eine bestimmte Zahl eine Dual-, Dezimal-, oder aber eine Hexadezimalzahl ist? Dual- und Hexadezimalzahlen besitzen als Kennzeichnung ein Sonderzeichen, das der Zahl vorangestellt wird. Dualzahlen werden mit dem Zeichen "%", Hexadezimalzahlen mit dem Zeichen "$" markiert.

| Dezimalzahlen | Dualzahlen | Hexadezimalzahlen |
|---|---|---|
| 0 | %00000000 | $0000 |
| 16 | %00010000 | $0010 |
| 255 | %11111111 | $00FF |

Sie brauchen übrigens nicht alle Hexadezimalzahlen auswendig zu lernen. Fast jeder Monitor kann unterschiedliche Zahlenformate umrechnen. Schauen Sie dazu bitte in die entsprechende Anleitung des Monitors.

## 4. Der Hauptspeicher

Jeder Computer besitzt einen Arbeits- oder auch Hauptspeicher (im Unterschied zu Massenspeichern wie Cassetten oder Disketten), der alle benötigten Daten und Programme aufnimmt. Wie und wo diese Speicherung vorgenommen wird, ist für uns uninteressant, wenn wir in BASIC programmieren.

Bei der Assembler-Programmierung ist es jedoch unumgänglich, in Grundzügen über den prinzipiellen Aufbau eines Hauptspeichers und vor allem über den Aufbau Ihres (!) Rechners Bescheid zu wissen.

## 4.1. Speicherorganisation

Um den Begriff Speicherorganisation zu klären, ist ein kurzer Ausflug in die BASIC-Programmierung angebracht. Wenn Sie - z.B. auf dem C64 - ein BASIC-Programm erstellen, wird dieses Programm mit allen Variablen, die es verwendet, im Rechnerspeicher abgelegt.

Aus dem bisher gesagten geht hervor, daß die CPU dieses Programm nicht versteht, BASIC ist eine Fremdsprache für unseren Rechner. Sie alle kennen den Begriff BASIC-Interpreter. Dieser Interpreter ist ein Assembler-Programm, das unsere BASIC-Befehle interpretiert und in Anweisungen der für die CPU verständlichen Assembler-Sprache umsetzt.

Das Assembler-Programm namens BASIC-Interpreter übernimmt auch die Aufgabe, unseren BASIC-Text in bestimmten Speicherbereichen abzulegen. Während der Duchführung eines BASIC-Programms werden fast immer Variablen benötigt (A=10, B%=23), denen Werte zugewiesen werden.

Wo diese Variablen gespeichert werden, ist wiederum dem BASIC-Interpreter überlassen und für den Benutzer uninteressant. BASIC ist eine sogenannte problemorientierte Sprache, im Gegensatz zur hardwareorientierten Sprache Assembler.

Da uns kein hilfreiches Programm zur Verfügung steht, das sich um die Speicherverwaltung kümmert, müssen wir selbst bei der Assembler-Programmierung angeben, wo unser Programm und die benötigten Variablen abzulegen sind.

Um einen geeigneten Ort zu bestimmen, müssen wir jedoch grundlegende Kenntnisse über die Speicherorganisation unseres Rechners besitzen.

Stellen Sie sich den Hauptspeicher eines Rechners bitte wie ein Hochhaus vor, das in einzelne Etagen unterteilt ist. Jede dieser Etagen ist eine Speicherzelle, in der man eine Zahl ablegen kann.

Jede Speicherzelle kann eine beliebige achstellige Dualzahl aufnehmen, also ein Byte (eine Zahl zwischen null und 255). Wie viele dieser Speicherzellen existieren, hängt vom verwendeten Rechner ab. Der C64 und der Plus/4 besitzen jeweils 65536 dieser Speicherzellen, der C16, C116 nur 16384, und der C128 mit dem größten Speicher verfügt über 131072 Speicherzellen.

Sie sehen, aufgrund der großen Anzahl der Speicherzellen ist die Maßeinheit Byte ziemlich unhandlich. Die Speicherkapazität eines Rechners wird üblicherweise in Kilobyte (Kb) gemessen (Ihr Gewicht messen Sie ebenfalls nicht in Gramm, sondern in Kilogramm).

Ein Kilobyte entspricht nun nicht - wie anzunehmen wäre - 1000, sondern 1024 Byte. Wenn Sie die obigen Angaben verschiedener Speicherkapazitäten durch 1024 teilen, erhalten Sie folgende Tabelle:

```
Speicherkapazitäten verschiedener Commodore-Heimcomputer

     C16, C116:    16 Kilobyte (=16384 Byte)
     C64, Plus/4:  64 Kilobyte (=65536 Byte)
     C128:        128 Kilobyte (=131072 Byte)
```

Jede dieser Speicherzellen kann mit einer beliebigen Zahl zwischen null und 255 - einem Byte - beschrieben werden. Um ein Byte in eine Speicherzelle zu schreiben, geben wir die Hausnummer (Etage in der Hochhaus-Analogie) der gewünschten Speicherzelle an, die sogenannte Adresse.

Die Adressierung der Speicherzellen beginnt mit der Adresse null. Die höchste Adresse hängt von der Speicherkapazität ab. Im C64 besitzt daher die letzte Speicherzelle die Adresse 65536.

Merken Sie sich bitte: der Hauptspeicher unseres Rechners ist in Speicherzellen unterteilt, die über ihre Adresse (Hausnummer) angesprochen werden und beliebige Zahlen zwischen null und 255 aufnehmen können.

## 4.2. Der Bildschirmspeicher

Nicht nur Assembler, sondern auch BASIC erlaubt es uns, beliebige Werte in eine angegebene Speicherzelle zu schreiben und zwar mit dem POKE-Befehl.

Wie Sie wissen, besitzt der POKE-Befehl das Format: POKE (ADRESSE),(ZAHL). Diesen POKE-Befehl werden wir nun verwenden, um ein maschinennahes Programm zu schreiben. Wir füllen den Bildschirm mit A's, ohne jedoch den PRINT-Befehl zu verwenden.

Da Sie alle BASIC beherrschen, kennen Sie sicherlich die Bildschirmcode-Tabelle, die sich im Handbuch Ihres Rechners eine Seite vor der ASCII-Tabelle befindet. Jedes Zeichen auf dem Bildschirm entspricht einer Zahl in den Speicherzellen des sogenannten Bildschirmspeichers oder Video-RAM's.

Der Bildschirmspeicher besteht aus 1000 Speicherzellen. Jede dieser Speicherzellen gehört zu einer bestimmten Bildschirmposition und enthält den Bildschirmcode des Zeichens, das sich an dieser Position befindet.

Dieser Bildschirmspeicher beginnt beim C64 mit der Adresse 1024. Die Speicherzelle mit der Hausnummer 1024 enthält den Bildschirmcode des Zeichens, das sich an der HOME-Position des Cursors befindet, also an Spalte eins von Zeile eins.

Der Bildschirmspeicher ist fortlaufend organisiert. Die Speicherzelle 1025 enthält daher den Code des Zeichens an der Position 2/1 (Spalte zwei/Zeile eins) und so weiter.

Mit diesem Wissen und dem POKE-Befehl können wir bereits beliebige Zeichen maschinennah auf dem Bildschirm ausgeben. Mit POKE 1024,(BILDSCHIRMCODE) verändern Sie das Zeichen an Position 1/1 (Spalte eins/Zeile eins).

Mit einer Schleife können wir den Bildschirm problemlos mit dem Zeichen A füllen.

```
100 VR=1024:REM STARTADRESSE VIDEO-RAM (C64)
110 FOR I=VR TO VR+1000
120 : POKE I,1:REM BILDSCHIRMCODE VON 'A'
130 NEXT
```

Besitzer eines C64 (und C128 im 64er-Modus) können dieses Programm sofort eingeben und starten. Es beschreibt alle 1000 Bildschirmpositionen mit dem Zeichen A, indem es in die zugehörigen Speicherzellen den Bildschirmcode von A schreibt.

Leider ist die Adresse dieses Bildschirmspeichers von Rechner zu Rechner unterschiedlich. Beim C16, C116 und Plus/4 beginnt das Video-Ram mit der Adresse 3072. Da die fortlaufende Organisation jedoch identisch ist, genügt es, Zeile 100 in

```
100 VR=3072:REM STARTADRESSE VIDEO-RAM (C16, PLUS4)
```

zu ändern, um das Programm auf dem C16 oder dem Plus/4 zu benutzen. Als einziger Commodore-Heimcomputer besitzt der C128 leider einen nicht direkt adressierbaren Bildschirmspeicher, d.h., es ist nicht ohne weiteres möglich, die Inhalte des Video-RAMs zu verändern (zumindest nicht im 80-Zeichen-Modus).

Aus diesem Grund bat ich alle C128-Besitzer, Ihren Rechner vorläufig im 64er-Modus zu betreiben. Sie werden später erfahren, wie es im 128er-Modus möglich ist, den Bildschirmspeicher dennoch anzusprechen, und zwar indirekt.

Zurück zu unserem Demoprogramm. Dieses Programm ist maschinennah, da es direkt bestimmte Speicherzellen anspricht. Es arbeitet analog zu unserem ersten echten Assembler-Programm, das - wie wir noch sehen werden - den Bildschirm auf die gleiche Art und Weise (Veränderung des Bildschirmspeichers) mit A's füllte.

## 4.3. Die Speicheraufteilung

Im letzten Kapitel sahen wir, wie der Speicher unseres Rechners organisiert ist (in Speicherzellen mit einer "Breite" von einem Byte) und wo sich der Bildschirmspeicher befindet.

Die Größe und Aufteilung des Speichers ist von Rechner zu Rechner (siehe Bildschirmspeicher) unterschiedlich. Wenn wir jedoch über die jeweilige Aufteilung Bescheid wissen, können wir jeden (!) Commodore-Rechner in Assembler programmieren, da die Befehle selbst identisch sind.

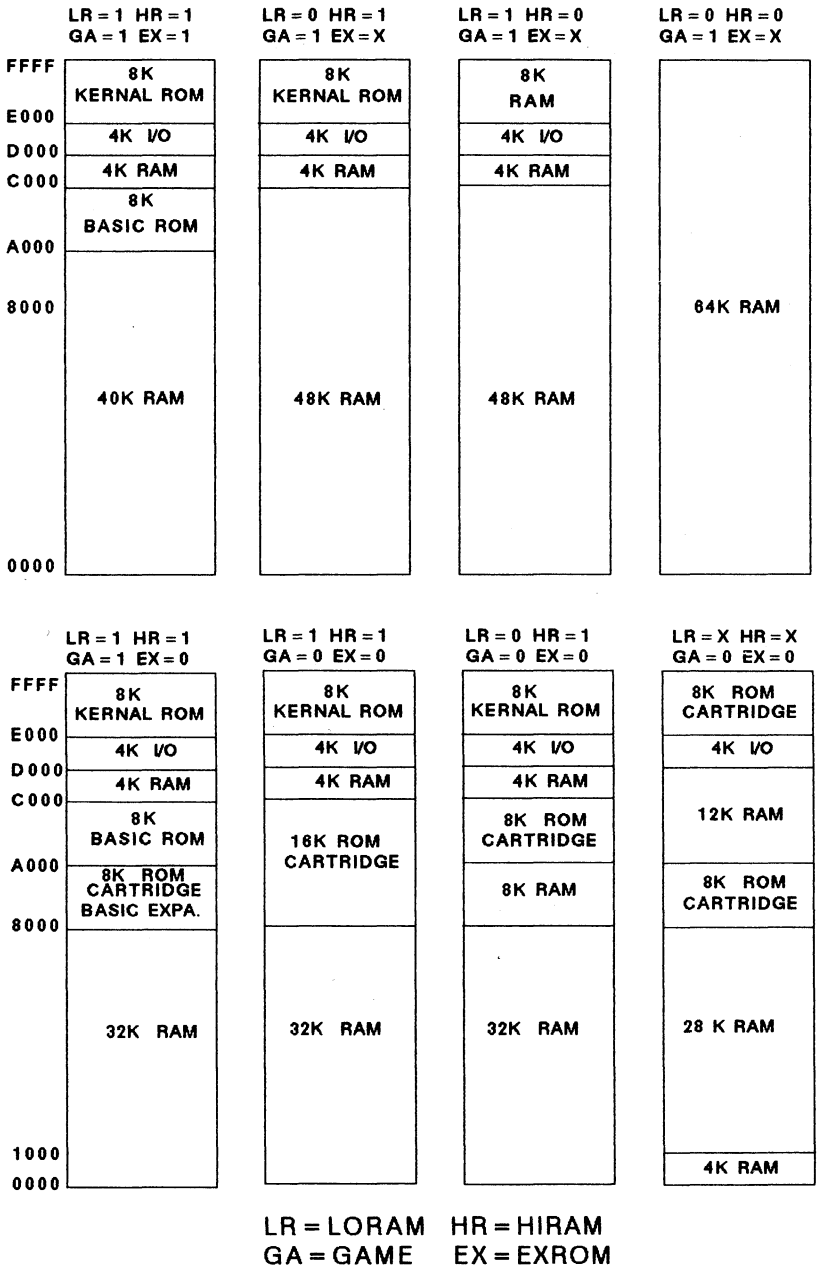



Abb. 6.5.1.2: Speicheraufteilung

Die Abbildung zeigt beispielhaft die Speicheraufteilung des Commodore C64. Folgende Bereiche sind zu unterscheiden:

### *1. Die Zeropage*

Eine Page oder Seite ist ein Speicherbereich, der 256 Speicherzellen umfaßt. Die erste Seite (Page null oder Zeropage) umfaßt den Bereich von Speicherzelle null bis Speicherzelle 255, die zweite Seite beginnt bei Speicherzelle 256 und endet bei Zelle 511 und so weiter. Man spricht auch vom Seitenkonzept der Speicheraufteilung (Unterteilung in Blöcke von je 256 Byte Umfang).

Die Zeropage ist für die Assembler-Programmierung außerordentlich wichtig. Verschiedene Assembler-Befehle können nur mit Hilfe dieses Speicherbereichs verwendet werden. In diesem speziellen Bereich befinden sich zudem für uns wichtige Informationen. Beispielsweise geben zwei dieser Speicherzellen immer Auskunft über die aktuelle Cursorposition (Spalte und Zeile).

### *2. Der Stack*

Der sogenannte Stack umfaßt ebenfalls eine Seite, d.h., 256 Speicherzellen. Er wird von der CPU benutzt, um sich beim Aufruf eines Unterprogramms zu merken, von wo der Aufruf kam, analog einem BASIC-Programm, bei dem das Programm nach dem Befehl RETURN mit jenem Befehl fortgesetzt wird, der dem Unterprogrammaufruf GOSUB folgt. Der Stack wird außerdem oft zur kurzzeitigen Speicherung von Daten verwendet, da sich der Rechner selbständig darum kümmert, wo (!) die Daten abgelegt werden und somit für uns jeglicher Verwaltungsaufwand entfällt.

### 3. *Der Bildschirmspeicher*

Der Bildschirmspeicher oder das Video-RAM besteht aus vier Seiten, also insgesamt 1024 Speicherzellen. Dieser Bereich enthält die Bildschirmcodes aller auf dem Bildschirm vorhandenen Zeichen.

Die erste Zelle des Bildschirmspeichers (Nummer 1024 beim C64) enthält den Bildschirmcode des Zeichens, das sich an der Position 1/1 (Spalte eins/Zeile eins) befindet, die zweite Zelle den Code des Zeichens an Position 2/1 (Spalte zwei/Zeile eins) und so weiter.

### 4. *Das BASIC-RAM*

Das BASIC-RAM ist jener Speicherbereich, in dem der BASIC-Interpreter Programme und Variablen ablegt.

### 5. *Das BASIC-ROM*

Im BASIC-ROM befindet sich der BASIC-Interpreter, jenes festeingebaute Programm, das unsere BASIC-Programme interpretiert und für ihre Ausführung sorgt. Ebenso wie alle anderen ROM-Bereiche können die Inhalte dieser Speicherzellen gelesen, jedoch nicht beschrieben und damit verändert werden.

### 6. *Das Zeichen-ROM*

Das Zeichen-ROM enthält den Zeichensatz unseres Rechners. d.h., alle verfügbaren Zeichen in Form des Punktmusters, aus dem sie aufgebaut sind.

*7. Das Kernal*

Das Kernal ist das sogenannte Betriebssystem des Rechners, ein wie der BASIC-Interpreter festeingebautes Assembler-Programm, das für alle Ein-/Ausgabeoperationen (Tastatur abfragen, Zeichen auf Drucker ausgeben etc.) zuständig ist. Ohne dieses Betriebssystem ist unser Rechner eine Ansammlung verschiedener elektronischer Bauteile, mit der wir nichts anfangen können.

Das Betriebssystem enthält viele für die Arbeit mit dem Rechner grundlegende Funktionen, die wir uns in späteren Kapiteln zunutze machen werden.

## 4.4. RAM und ROM

Wie diese Unterteilung zeigt, unterscheidet man RAM- und ROM-Speichereinheiten. Die ROM-Speicherzellen (ROM = read only memory, nur lesbare Speichereinheit) können nicht mit beliebigen Werten beschrieben werden. Sie enthalten fest "eingebrannte" Daten oder Programme, die auch dann erhalten bleiben, wenn der Rechner ausgeschaltet wird.

RAM-Speicherzellen (RAM = random access memory, Speicher mit wahlfreiem (Lesen oder Schreiben) Zugriff) verlieren ihre Inhalte sofort, wenn die Stromversorgung unterbrochen wird. Im Gegensatz zum ROM kann der Inhalt von RAM-Speicherzellen jedoch beliebig verändert werden.

Aus diesem Grund ist der RAM-Speicher für uns der interessantere von beiden. Im RAM-Speicher werden wir Assembler-Programme und benötigte Daten ablegen.

Eines der Grundprobleme bei der Assembler-Programmierung ist die Suche nach freien Speicherbereichen, die wir für unsere Programme nutzen können. In mehreren Bereichen des Speichers werden ständig benötigte Daten abgelegt, die wir nicht einfach überschreiben dürfen.

So ist z.B: die Zeropage für uns vorläufig tabu. Wie erläutert, enthält sie Informationen, die das Betriebssystem zu seiner Arbeit benötigt und die wir nicht einfach verändern dürfen.

Ein gut geeigneter Speicherbereich zur Ablage von Assembler-Programmen ist beim C64 der Bereich 49152 bis 53247, der normalerweise unbenutzt ist. In diesem vier Kilobyte (4096 Byte) großen Bereich werden wir (beim C64) unsere Assembler-Programme ablegen. Im folgenden werde ich zur Adressangabe das Hexadezimalsystem benutzen. Hexadezimal ausgedrückt beginnt der Bereich bei Adresse $C000 und endet bei $CFFF.

Beim C16, C116 und Plus/4 werden wir den Bereich $03F7-$0436 verwenden, der 64 Byte groß ist, den sogenannten RS-232-Input-Puffer. Diesen Puffer verwendet Ihr Rechner zum Betrieb der RS-232-Schnittstelle, die für spezielle Arten der Datenübertragung - z.B. den Betrieb eines Telefon-Modems - verwendet wird und normalerweise unbenutzt ist.

C16, C116- und Plus/4-Besitzer sollten daher im folgenden die Angabe $C000 immer durch $03F7 ersetzen.

## 5. Programmieren mit dem Monitor

Wie erläutert, setzt dieses Buch die Verwendung eines Monitors zur Programmeingabe voraus. Der folgende Abschnitt zeigt, wie Programme mit einem Monitor eingegeben, editiert, gespeichert und geladen werden.

Die Unterschiede im Format der einzelnen Monitor-Befehle sind von Monitor zu Monitor äußerst gering, so daß sich dieser Abschnitt gleichermaßen auf den eingebauten Monitor des C16, Plus/4 und C128, auf den im Anhang abgebildeten oder einen beliebigen anderen Monitor bezieht.

## 5.1. Laden und Starten eines Monitor-Programms

C16, Plus/4- und C128-Benutzer besitzen einen fest im Rechner eingebauten Monitor, der mit dem Befehl MONITOR aufgerufen und mit X verlassen wird.

C64-Besitzer benutzen entweder den in diesem Buch (Anhang) abgebildeten Monitor oder einen der vielen in diversen Fachzeitschriften abgedruckten. Manche für den C64 erhältlichen Monitore müssen mit einem Befehl wie LOAD"NAME),8,1 geladen werden statt mit LOAD"NAME",8. Wichtig ist bei diesen Monitoren die Eingabe des Befehls NEW vor dem Starten des Monitors, da Sie ansonsten Schwierigkeiten mit eventuell einzugebenden BASIC-Programmen bekommen.

Wenn Sie einen Monitor erwerben oder abtippen, achten Sie bitte darauf, daß sich dieser nicht im Bereich $C000-$CFFF befindet (siehe Anleitung), da wir diesen Bereich im folgenden zur Ablage unserer Programme verwenden.

## 5.2. Assembler-Programme mit einem Monitor eingeben

Jeder Monitor besitzt verschiedene Befehle, die uns bei der Erstellung eines Programms unterstützen. Um Assembler-Befehle mit einem Monitor einzugeben, verwenden wir den Befehl Assemble, der bei jedem Monitor mit dem Buchstaben A abgekürzt wird. Dem Assemble-Befehl folgt die hexadezimale Angabe einer Adresse. Der folgende Assembler-Befehl wird an eben dieser Adresse vom Monitor im Speicher abgelegt.

Das allgemeine Format lautet:

```
A (STARTADRESSE) (ASSEMBLER-BEFEHL)
```

(Denken Sie bitte an den Unterschied zum abgedruckten Monitor. Geben Sie bei ihm bitte statt des A ein "," ein.)
Der hinten abgedruckte Monitor und einige andere für den C64 erlauben Eingaben nur ohne A. Geben Sie hier: Komma Adresse Befehl ein.

Beispiel: , C000 LDA # $ 00

Zwischen der Adresse und dem Befehl werden von diesem Monitor keine Zahlen ausgegeben.

Monitore erwarten, wie erläutert, Zahleneingaben im Hexadezimalsystem. Da Startadressen immer (!) hexadezimal anzugeben sind, können Sie bei den meisten Monitoren auf die Eingabe des Zeichens "$" vor der Startadresse verzichten.

Der im C16, C116, Plus/4 und C128 eingebaute Monitor verlangt sogar ausdrücklich nach Eingabe der Startadresse ohne das Zeichen "$". Der Befehl A $03F7 LDA #$01 wird vom Monitor nicht akzeptiert und muß durch A 03F7 LDA #$01 ersetzt werden.

Jeder Assembler-Befehl besitzt eine bestimmte Länge, der Befehl LDA #$01 z.B. eine Länge von zwei Byte. Diesen Befehl legt der Monitor in den Speicherzellen $C000 und $C001 ab, wenn wir als Startadresse C000 angeben (A C000 LDA #$01). Die Adresse, ab der der nächste Befehl abgelegt wird, steht nun fest ($C000 + $02 = $C002) und muß vom Benutzer nicht mehr eingegeben werden.

Sobald Sie einmalig einen Befehl mit einer Startadresse eingeben, wird Ihnen der Monitor daher die Adresse des jeweils folgenden Befehls vorgeben und Sie müssen nur noch den Befehl selbst eintippen (siehe AUTO-Befehl in BASIC).

Merken Sie sich bitte: ein Assembler-Programm wird mit einem Monitor eingegeben, indem der erste Befehl mit der gewünschten Startadresse des Programms eingegeben wird. Bei allen folgenden Befehlen wird die jeweilige Befehlsadresse vom Monitor vorgegeben und es genügt vollkommen, den Befehl selbst einzugeben.

Üben Sie diese Programmeingabe bitte mit dem folgenden Programm:

Rahmenfarben ändern (C64)

```
A C000 LDX #$00
TXA
STA $D020
DEX
JMP $C002
```

Rahmenfarben ändern (C16, C116, Plus/4)

```
A 03F7 LDX #$00
TXA
STA $FF19
DEX
JMP $03F7
```

Verlassen Sie den Monitor anschließend mit dem Befehl X und starten Sie das Assembler-Programm mit SYS 49152 (C64) bzw. SYS 1015 (C16, C116 und Plus/4), d.h., mit dem SYS-Befehl und der Angabe der Programm-Startadresse.

Haben sie das Programm korrekt eingegeben, ergibt sich ein recht netter Effekt. Der Bildschirmrahmen wird von Wellen überzogen. Das Programm befindet sich in einer Endlosschleife, die nur mit STOP+RESTORE (C64) oder durch Drücken des RESET-Knopfes (C16, C116, Plus/4) verlassen werden kann.

Fehler bei der Eingabe korrigieren Sie, indem Sie den betreffenden Befehl übertippen und RETURN drücken, analog zur Änderung einer BASIC-Befehlszeile.

In den folgenden Kapiteln werden wir Assembler-Programme eingeben, die nicht mehr auf den Bildschirm passen. Zur Durchsicht und Korrektur des Programms benötigen wir daher ein Äquivalent zum LIST-Befehl.

Dieser Monitor-Befehl nennt sich Disassemble und wird mit dem Buchstaben D abgekürzt. Das allgemeine Format lautet (vergleiche LIST (VON)-(BIS)):

```
D (STARTADRESSE) (ENDADRESSE)
```

Der im C16, C116, Plus/4 und C128 eingebaute Monitor bietet als besonderen Komfort das automatische Listen mehrerer Befehle ab der angegebenen Startadresse.

Zum Listen unseres Demoprogramms genügt die Eingabe D 03F7, um alle Befehle auf dem Bildschirm darzustellen. Die beim C64 üblichen Monitor-Programme bieten diesen Komfort normalerweise nicht, so daß die Eingabe einer Endadresse unumgänglich ist. Das Programm besteht aus fünf Befehlen. Rechnen Sie mit einer durchschnittlichen Befehlslänge von zwei Byte. Das Beispielprogramm ist somit etwa 10 Byte lang.

Geben Sie zum Listen des Programms auf dem C64 daher ein:

```
D C000 C00A
```

Mit dem Disassemble-Befehl ist es jederzeit möglich, auch umfangreiche Programme zu listen und zu korrigieren. Ob bei der hexadezimalen Angabe des zu listenden Speicherbereichs das Zeichen "$" benötigt wird oder entfallen kann, ist von Monitor zu Monitor unterschiedlich.

Wenn Sie bei der Disassemblierung eines Speicherbereichs seltsame Assembler-Befehle oder gar Fragezeichen im Anschluß an das eigentliche Programm entdecken, stören Sie sich bitte nicht daran. Hinter unserem Programm befinden sich ja ebenfalls Speicherzellen, die (unbestimmte) Werte enthalten. Der Monitor versucht, diese Werte als Assembler-Befehle zu interpretieren. Existiert zum Inhalt einer Speicherzelle kein zugehöriger Assembler-Befehl, gibt der Monitor ein Fragezeichen aus.

## 5.3. Speicherung eines Assembler-Programms im Rechner

Bei der Eingabe des letzten Demoprogramms stellten Sie fest, daß der Monitor unmittelbar nach der Eingabe eines Befehls die gesamte Befehlszeile in einem anderem Format darstellt. Der Befehlsadresse folgen mehrere zweistellige Hexadezimalzahlen ohne das Kennzeichen "$" (zumindest bei den meisten Monitoren), die jedoch an Ziffern wie A,B,C,D,E oder F zu erkennen sind. Diesen Hexadezimalzahlen folgt der von Ihnen eingegebene Assembler-Befehl.

(Beim hinten abgedruckten Monitor werden diese Zahlen nicht ausgegeben.)

Rahmenfarben ändern (C64)

```
A C000 A2 00      LDX #$00
A C002 8A    TXA
A C003 8D 20 D0     STA $D020
A C006 CA    DEX
A C00A 4C 02 C0     JMP $C002
```

Rahmenfarben ändern (C16, C116 und Plus/4)

```
A 03F7 A2 00 LDX #$00
A 03F9 8A    TXA
A 03FA 8D 19 FF     STA $FF19
A 03FD CA    DEX
A 03FE 4C F9 03     JMP $03F9
```

Da Ihr Rechner prinzipiell nur Zahlen verarbeiten kann, wird auch ein Programm in Zahlenform codiert im Speicher abgelegt. Jedem Assembler-Befehl entspricht eine Zahl zwischen null und 255, ein Byte. Dieses Byte stellt der Monitor unmittelbar hinter der Adresse des Befehls in hexadezimaler Form dar. A2 (genauer: $A2) entspricht dem Befehl LDX, 8A dem Befehl TXA und so weiter.

(Sie brauchen diese Zahlen und die zugehörigen Befehle aber nicht auswendigzulernen, denn Sie haben ja einen Monitor, der

die Befehle in Zahlen und die Zahlen in die zugehörigen Befehle umrechnet.)

Die meisten Befehle, die wir kennenlernen, benötigen ein Argument, vergleichbar BASIC-Befehlen wie PEEK(ARGUMENT) oder INT(ARGUMENT). Das Argument ist eine ein oder zwei Byte lange Zahl. Je nach Länge von Assembler-Befehl plus Argument spricht man von einem Ein-, Zwei- oder Drei-Byte-Befehl.

Dem Befehl LDX folgt das Argument 00 ($00). LDX #$00 ist daher ein Zwei-Byte-Befehl, JMP $C002 (4C 02 C0) ein Drei-Byte-Befehl mit dem Argument 02 C0.

Wenn Sie sich die zu den Befehlen gehörenden Zahlen mit dem hinten abgedruckten Monitor anschauen möchten, so benutzen Sie dazu bitte den Memory-Befehl M. Um beispielsweise die Zahlen zu sehen, die zu

```
,C000 LDA $0300
```

gehören, geben Sie M C000 sowie RETURN ein und Sie erhalten die Anzeige:

```
:C000 AD 00 03 .....
```

Um zu verdeutlichen, wozu Argumente benötigt werden: der Befehl JMP ist ein Sprungbefehl, vergleichbar dem BASIC-Befehl GOTO, und setzt wie dieser die Programmverarbeitung an der angegebenen Stelle fort.

Das Sprungziel wird jedoch nicht in Form einer Zeilennummer, sondern einer Befehls-Adresse angegeben. Der Sprung erfolgt zur angegebenen Speicherzelle und das Programm wird mit dem in dieser Zelle enthaltenen Befehl fortgesetzt, in diesem Fall mit dem Befehl an Adresse $C002 (TXA).

Die Adresse $C002 ist das Argument des Befehls JMP. Der Befehl JMP wird in Speicherzelle $C00A abgelegt. Die Hexadezimalzahl $C002 wird in zwei (!) Speicherzellen abgelegt. Der

Grund: eine Speicherzelle kann nur Zahlen zwischen null und 255 enthalten, also ein Byte. Zur Darstellung einer größeren Zahl werden mehrere Bytes benötigt.

Mit zwei Bytes kann eine beliebige Zahl zwischen null und 65535 dargestellt werden. Verständlich wird die Aufteilung einer großen Zahl in zwei Bytes jedoch erst, wenn wir uns die Hexadezimal-Darstellung von Zahlen betrachten.

Die Zahl 49154 wird hexadezimal in der Form $C002 dargestellt. Diese vierstellige hexadezimale Zahl wird in zwei Abschnitte unterteilt. Abschnitt eins enthält die ersten beiden Ziffern (C0), Abschnitt zwei die beiden letzten Ziffern (02).

Jeder dieser Abschnitte wird als eigene Zahl zwischen null ($00) und 255 ($FF) betrachtet. Die vierstellige Hexadezimalzahl wird in zwei Bytes unterteilt. Man spricht vom höherwertigen oder High-Byte (die beiden linken Ziffern) und niederwertigen oder Low-Byte (die beiden rechten Ziffern).

Diese beiden Bytes werden in zwei aufeinanderfolgenden Speicherzellen untergebracht und zwar bei allen Commodore-Computern in der etwas gewöhnungsbedürftigen Form Low-Byte/High-Byte, dem sogenannten Adreßformat. In der ersten Speicherzelle wird die niederwertige, in der folgenden Speicherzelle die höherwertige Hälfte der Zahl untergebracht.

Diese Art der Adreß-Speicherung erklärt die Darstellung des Drei-Byte-Befehls JMP $C002 durch die drei Bytes $4C $02 $C0. Das Byte $4C ist der Befehls-Code des JMP-Befehls. $02 $C0 ist die Darstellung der Adresse $C002 in der Form Low-Byte/High-Byte.

Merken Sie sich diese Art der Adreß-Darstellung unbedingt. Wir werden viele Befehle kennenlernen, die mit Zwei-Byte-Adressen arbeiten. Bei allen diesen Adressen wird das niederwertige vor (!) dem höherwertigen Byte gespeichert.

Aber keine Sorge: Durch unsere vielen Übungen werden Sie das schnell lernen und der Monitor nimmt Ihnen diese Arbeit sowieso ab.

## 5.4. Programme speichern und laden

Assembler-Programme nützen uns recht wenig, wenn sie beim Ausschalten des Rechners jedesmal verlorengehen. Mit den BASIC-Befehlen SAVE und LOAD können wir jedoch nur BASIC- und keine Assembler-Programme speichern.

Jeder Monitor besitzt jedoch zwei gleichwertige Befehle, S (für SAVE) und L (für LOAD). Der Befehl S besitzt das Format:

```
S"(NAME)",(GERÄTENR.),(STARTADRESSE),(ENDADRESSE+1)
```

Um ein Assemblerprogramm zu speichern, müssen Sie ebenso wie bei einem BASIC-Programm einen Dateinamen (NAME) angeben. Anschließend folgt die (GERÄTENR.). Wollen Sie das Programm auf Cassette speichern, geben Sie 01 ein, zum Speichern auf Diskette hingegen die Gerätenummer 08.

Vielleicht wundern Sie sich, warum die Eingabe 1 oder 8 nicht genügt. Dies liegt an der bereits erwähnten Eigenheit von Monitoren, auf hexadezimalen Eingaben zu bestehen. 8 entspricht der zweistelligen Hexadezimalzahl $08 und 1 der Zahl $01 (wobei das Dollarzeichen jedoch nicht mit eingegeben werden darf).

Nun geben Sie an, wo sich das abzuspeichernde Programm befindet und wo es endet. Der Monitor wird genau diesen Speicherbereich auf Cassette bzw. Diskette schreiben. Beide Angaben erfolgen wiederum hexadezimal. Ein Beispiel:

```
A C000 EE 20 D0    INC $D020
A C003 4C 00 C0    JMP $C000
A C006
```

Dieses kurze Programm verändert ununterbrochen die Farben des Bildschirmrahmens. Es beginnt bei der im Assemble-Befehl angegebenen Startadresse $C000 und endet bei Adresse $C005, ein Byte vor der nächsten Befehlsadresse $C006.

Beim Speichern können Sie auf das Dollarzeichen bei der Angabe der Start- und der Endadresse verzichten. Beachten Sie jedoch, daß dem Monitor nicht die echte Endadresse, sondern die (ENDADRESSE+1) anzugeben ist. Im Beispiel wäre dies die Adresse des nächsten Befehls, $C006.

Speichern auf Cassette: `S"TESTPROG",01,C000,C006`
Speichern auf Diskette: `S"TESTPROG",08,C000,C006`

Beide Befehle speichern das Programm unter dem Namen TESTPROG auf Cassette bzw. Diskette. Manche Monitore verlangen übrigens beim Speichern (und Laden) ein Leerzeichen anstelle eines Kommas. Welches dieser beiden Trennzeichen Ihr Monitor erwartet, müssen Sie ausprobieren (eingebauter Monitor im C16, C116, Plus/4 und C128: Komma).

Der Befehl L besitzt folgendes Format:

```
L"(NAME)",(GERÄTENR.)
```

Laden von Cassette: `L"TESTPROG",01`
Laden von Diskette: `L"TESTPROG",08`

Die Angabe der Start- und Endadresse entfällt beim Laden. Das Programm TESTPROG wird automatisch nach $C000 geladen, in jenen Bereich, an dem es sich beim Speichern befand.

Beim C16, C116 und Plus/4 geben Sie bitte das gleiche Programm (aber mit Adresse $FF19 statt $D020), jedoch ab Adresse $03F7 ein:

```
A 03F7 EE 20 D0    INC $FF19
A 03FA 4C 00 C0    JMP $03F7
A 03FD
```

Sie laden das Programm wie beschrieben, geben beim Speichern jedoch die für Sie gültige Start- und Endadresse(+1) ein.

Speichern auf Cassette: `S"TESTPROG",01,03F7,03FD`
Speichern auf Diskette: `S"TESTPROG",08,03F7,03FD`

Zur Übung empfehle ich Ihnen, den Bildschirminhalt zu speichern und - nach dem Löschen des Bildschirms - wieder zu laden. Beim C64 müssen Sie den Bereich $0400-$07FF speichern und beim C16, Plus/4 den Bereich $0C00-$0FFF. Beachten Sie, daß die Endadresse plus eins (!) anzugeben ist.

## 6. Befehlsbearbeitung durch die CPU

Die CPU besitzt einen Programmzähler oder program counter. Der Programmzähler weist immer auf die Adresse, an der sich der nächste zu bearbeitende Befehl befindet.

Da der gesamte Rechnerspeicher addressiert werden muß ($0000-$FFFF beim C64), ist diese Adresse zwei Byte lang (siehe oben). Der Programmzähler besteht aus zwei Registern, speziellen Speicherzellen, die wie üblich jeweils ein Byte aufnehmen können und die nieder- bzw. die höherwertige Hälfte der jeweiligen Adresse enthalten.

Nehmen wir an, der Programmzähler weist auf die Adresse $C000, an der sich der erste Befehl unseres Programms befindet (LDX #$00). Über den sogenannten Datenbus wird der Inhalt der Speicherzelle $C000 geholt, die Zahl $A2.

Die Zahl wird nun decodiert und als Befehl LDX erkannt. Anschließend wird der Programmzähler um eins erhöht und weist danach auf die Adresse $C001. Das Argument des Befehls LDX, die Zahl $00 befindet sich an dieser Adresse und wird ebenfalls mit Hilfe des Datenbusses vom Rechnerspeicher in die CPU übertragen.

Ob ein Befehls-Argument ein oder aber zwei Byte umfaßt, erkennt die CPU am Befehlscode. Nachdem das Befehlsargument vom Speicher zur CPU transportiert wurde, wird der Befehl ausgeführt.

Der Programmzähler wird bei jeder Übertragung eines Bytes um den Wert eins erhöht und weist somit nach der Bearbeitung eines Befehls (und seines Argumentes) auf den folgenden Befehl, der auf die gleiche Weise abgearbeitet wird.

## 7. Die CPU-Register

Im vorigen Abschnitt lernten wir zwei CPU-Register kennen, die zusammen den Programmzähler bilden. Die verschiedenen CPU-Register sind die wichtigsten Speicherzellen, die unser Rechner besitzt.

### *1. Der Akkumulator*

Das wichtigste CPU-Register ist der sogenannte Akkumulator. Fast alle Operationen sind nur mit Hilfe des Akkumulators möglich. Um beispielsweise eine bestimmte Speicherzelle mit einem beliebigen Wert zu beschreiben (analog dem POKE-Befehl), muß dieser Wert zuerst in den Akkumulator geladen werden. Anschließend kann der Akkumulatorinhalt in die gewünschte Speicherzelle übertragen werden.

Außer für den Datentransfer ist der Akkumulator für Berechnungen aller Art zuständig. Additionen oder Subtraktionen können nur in diesem speziellen Register stattfinden.

*2. Die Indexregister*

Die CPU besitzt zwei Indexregister, die als X-Register und Y-Register bezeichnet werden. Beide Register werden vorwiegend als Schleifenzähler und zur Bearbeitung von Tabellen verwendet.

*3. Das Status-Register*

Das Statusregister enthält wie alle übrigen Register ein Byte. Dieses Byte gibt über den Prozessor-Status Auskunft, z.B. über das Ergebnis von Subtraktionen oder Additionen. Am Inhalt des Statusregisters können wir z.B. erkennen, ob das Ergebnis einer Addition größer oder kleiner als 255 ist oder ob eine Programmschleife beendet ist.

*4. Der Stapelzeiger*

Der Stapel ist ein spezieller, eine Seite (256 Byte) langer Speicherbereich, der zur kurzfristigen Ablage von Informationen geeignet ist und mit Hilfe des Stapelzeigers verwaltet wird.

## 8. Die Adressierungsarten

Sie wissen bereits von BASIC-Programmen, daß es möglich ist, Variablen auf verschiedene Weise anzusprechen, zu adressieren. Bei der direkten Adressierung geben Sie den Namen einer einfachen Variablen an, z.B. PRINT A$.

Die indirekte Adressierung verwendet Arrayvariablen. Welche Variable angesprochen wird, bestimmt der Wert des angegebenen Index (PRINT A$(1):PRINT B%(2)). Auf die Variable wird indirekt zugegriffen.

Bei der Assembler-Programmierung steht uns eine Vielzahl verschiedener Addressierungsarten zur Verfügung, deren effektiver Einsatz ungemein wichtig ist.

Daher werde ich darauf verzichten, Ihnen sofort zu Beginn unseres Assembler-Kurses eine Vielzahl von Befehlen zu präsentieren. Das Wissen um die verschiedenen Adressierungsarten ist weitaus wichtiger als die Kenntnis auch des allerletzten Befehls.

Im folgenden werden zwei grundlegende Assembler-Befehle erläutert, LDA und STA. Mit diesen Befehlen werden die meisten Adressierungsarten durchgespielt. In den folgenden Kapiteln werden nach und nach weitere Befehle eingeführt, an denen zusätzliche Adressierungsmöglichkeiten demonstriert werden.

## 8.1. LDA und STA

Die Assembler-Programmierung besteht vorwiegend darin, Inhalte bestimmter Speicherzellen zu lesen oder zu verändern. Zu diesem Zweck gibt es die Klasse der Transferbefehle, die für den Datentransport zuständig sind.

Die grundlegendsten und am häufigsten verwendeten Transferbefehle sind LDA und STA. Die Kürzel zur Bezeichnung von Assembler-Befehlen, die immer aus drei Buchstaben bestehen, werden als Mnemonics bezeichnet und sind so gewählt, daß die Funktion meist direkt aus dem Namen hervorgeht. Die Abkürzung LDA steht für load accumulator, also lade den Akkumulator (mit einem Wert), die Abkürzung STA für store accumulator, also übertrage den Inhalt des Akkumulators (in eine Speicherzelle).

Mit diesen grundlegenden Transferbefehlen ist es möglich, den Inhalt beliebiger Speicherzellen zu lesen und zu verändern. Zu diesem Zweck existieren verschiedene Adressierungsarten, die im folgenden besprochen werden.

## 8.2. Unmittelbare und absolute Adressierung

Bei der einfachsten Adressierungsart, der unmittelbaren Adressierung, folgt dem Befehl als Argument ein - üblicherweise hexadezimal angegebener - Wert.

Dem Befehl LDA ("lade den Akkumulator") folgt der Wert, mit dem der Akkumulator geladen werden soll. Da der Akkumulator ein Register mit einer "Breite" von acht Bit (einem Byte) ist, ist das Argument eine Zahl zwischen null und 255. Um die unmittelbare Adressierung von anderen Adressierungsarten zu unterscheiden, befindet sich vor dem eigentlichen Wert das Zeichen "#".

Beispiele:

1. LDA #$00: Akkumulator mit dem Wert $00 (=dezimal null) laden.
2. LDA #$80: Akkumulator mit dem Wert $80 (=dezimal 128) laden.
3. LDA #$FF: Akkumulator mit dem Wert $FF (=dezimal 255) laden.

Befehle, die mit unmittelbarer Adressierung arbeiten, sind immer Zwei-Byte-Befehle. Der Befehl LDA #$FF besteht aus einem Byte, das den Befehlscode selbst enthält, und einem weiteren Byte, das das Argument $FF enthält.

Erste sinnvolle Programme lassen sich durch Kombination der Befehle LDA und STA erstellen. Mit STA ("übertrage den Inhalt des Akkumulators") wird der aktuelle Akkumulator-Inhalt in die angegebene Speicherzelle kopiert.

Da dem Befehl STA kein Wert, sondern die Adresse einer Speicherzelle als Argument angegeben wird, ist die unmittelbare Adressierung mit STA nicht möglich.

Die einfachste Adressierungart, die STA kennt, ist die absolute Adressierung, bei der als Argument eine absolute Speicheradresse angegeben wird.

Beispiele:

1. STA $C000: Akkumulator-Inhalt in Speicherzelle $C000 (=dezimal 49152) übertragen.

2. STA $C002: Akkumulator-Inhalt in Speicherzelle $C002 (=dezimal 49154) übertragen.

Da dem Befehlscode des Wortes STA eine Zwei-Byte-Adresse folgt, handelt es sich bei allen Beispielen um Drei-Byte-Befehle.

Mit dem folgenden Programm können Sie die Wirkungsweise der unmittelbaren und der absoluten Adressierung erproben.

```
Unmittelbare und absolute Adressierung (C64)

        A C000 A9 01            LDA #$01
        A C002 8D 0A C0         STA $C00A
        A C005 00               BRK

Unmittelbare und absolute Adressierung (C16, Plus/4)

        A 03F7 A9 01            LDA #$01
        A 03F9 8D FF 03         STA $03FF
        A 03FC 00               BRK
```

Dieses Programm demonstriert zugleich einige bisher unbekannte Möglichkeiten von Monitoren. Starten Sie es bitte nicht (!) auf die übliche Art und Weise (Monitor mit X verlassen, Programm von BASIC aus mit SYS 49152 bzw. SYS 1015 aufrufen).

Jeder Monitor besitzt einen Befehl zum Starten eines Assembler-Programms namens GO, der mit dem Buchstaben G abgekürzt wird. Dem Befehl selbst folgt die Startadresse des jeweiligen Programms:

```
G (STARTADRESSE)
```

Starten Sie das eingegebene Programm mit G C000 bzw. (C16, Plus/4) mit G 03F7. Das Programm arbeitet wie folgt (Befehle für C16, Plus/4 in Klammmern):

1. Der Befehl LDA #$01 lädt den Akkumulator mit dem unmittelbar angegebenen Wert $01.

2. Der Befehl STA $C00A (STA $03FF) schreibt den momentanen Inhalt des Akkumulators ($01) in die absolut angegebene Speicherzelle $C00A ($03FF).
3. Der bisher unbekannte Assembler-Befehl BRK unterbricht die Bearbeitung des laufenden Programms (analog dem BASIC-Befehl STOP) und bewirkt die Rückkehr in den Monitor.

Ob die beiden Befehle tatsächlich wie gewünscht ausgeführt wurden, können Sie mit dem Monitor-Befehl MEMORY (Abkürzung M) überprüfen.

```
M (STARTADRESSE) (ENDADRESSE)
```

Der MEMORY-Befehl zeigt beliebige Speicherbereiche an. Die Inhalte der betreffenden Speicherzellen werden in hexadezimaler Form ausgegeben. Viele Monitore - z.B. der im C16, Plus/4 und C128 eingebaute - geben diese Inhalte zusätzlich am rechten Bildschirmrand in ASCII-Form aus.

Ebenso wie beim BASIC-Befehl LIST ist die Angabe der Endadresse optional. Wird nur die Anfangsadresse eingegeben, gibt der Monitor eine Bildschirmzeile aus, die die Inhalte der ersten acht Speicherzellen ab der angegebenen Adresse wiedergibt.

Um den Inhalt von Speicherzelle $C00A ($03FF) zu überprüfen, geben Sie je nach Rechner ein:

```
M C00A (beim C64)
M 03FF (beim C16 und Plus/4)
```

Die erste ausgegebene Hexadezimalzahl entspricht dem Inhalt der angegebenen Speicherzelle. Wurde das Programm korrekt eingegeben, sollte diese Speicherzelle den Wert $01 besitzen.

Die meisten Monitor-Programme verzichten übrigens auf die Ausgabe des Zeichens "$" vor den einzelnen Werten, da die hexadezimale Form wie erläutert von Monitoren meist als selbstverständlich vorausgesetzt wird.

Das Programm zeigt zugleich weitere noch unbekannte Eigenschaften von Monitoren. Nach dem Start des Monitors und der Rückkehr aus einem Assembler-Programm (BRK-Befehl) sehen Sie auf dem Bildschirm die sogenannte Registeranzeige.

Nach der Ausführung unseres Programms könnten die Register beispielsweise die folgenden Werte enthalten:

```
PC   IRQ  SR AC XR YR SP
C006 EA31 30 01 00 01 F2
```

- PC : Programmzähler
- IRQ : Zeiger auf Systeminterrupt (IRQ)
- SR : Status-Register
- AC : Akkumulator
- XR : X-Register
- YR : Y-Register
- SP : Stack-Pointer

Diese Register-Anzeige gibt Auskunft über die momentan in den erwähnten Registern enthaltenen Werte. Zusätzlich erhalten wir Auskunft über den IRQ und den Stack-Pointer, die uns jedoch vorläufig nicht weiter interessieren.

Wichtig für uns ist die Anzeige des Akkumulator-Inhalts. Der Akkumulator enthält nach dem BRK-Befehl den Wert $01, gemäß dem Befehl LDA #$01 ("lade den Akkumulator mit dem Wert $01"). Wie wir sehen, wurde der Inhalt des Akkumulators durch den folgenden Befehl STA $C00A (STA $03FF) nicht beeinflußt.

Transferbefehle, die den Inhalt einer Speicherzelle oder auch eines Registers übertragen, verändern niemals (!) den Originalwert. Ein Befehl wie STA kopiert den Inhalt des Akkumulators, der unverändert erhalten bleibt!

Mit diesen beiden Befehlen, LDA und STA, können wir bereits eine Menge anfangen. Zusammengenommen entsprechen Sie dem POKE-Befehl, der ebenfalls eine beliebige Zahl (zwischen null und 255) in eine angegebene Speicherzelle kopiert.

Mit LDA und STA können wir z.B: Zeichen auf dem Bildschirm ausgeben, indem wir den Bildschirmcode des gewünschten Zeichens mit LDA in den Akkumulator laden und den Akkumulatorinhalt mit STA in eine Speicherzelle des Video-RAM's kopieren.

Als weiteres Demoprogramm bietet sich die Veränderung der Farben des Bildschirmrahmens an. Alle Commodore-Computer besitzen ein elektronisches Bauteil, einen Chip, der für die Bildschirmsteuerung zuständig ist. Dieser Chip (beim C64 TED, beim C16, Plus/4 und C128 VIC genannt) verfügt über spezielle Register - also Speicherzellen -, die wesentliche Parameter enthalten, z.B. die aktuelle Farbe des Bildschirmrahmens.

Beim C64 besitzt dieses spezielle Register die Hausnummer $D020, beim C16 und Plus/4 die Adresse $FF19. Durch Verändern des Registerinhaltes können wir die Rahmenfarbe beliebig ändern.

Rahmenfarbe ändern (C64)

```
A C000 A9 01            LDA #$01
A C002 8D 20 0D         STA $D020
A C005 00               BRK
```

Rahmenfarbe ändern (C16, Plus/4)

```
A 03F7 A9 01            LDA #$01
A 03F9 8D 19 FF         STA $FF19
A 03FC 00               BRK
```

Starten Sie das Programm wie zuvor mit G C000 (C64) bzw. G 03F7 (C16, Plus/4). Die Rahmenfarbe sollte sich ändern, es sei denn, die mit dem Wert $01 angegebene Farbe entspricht exakt jener, die der Rahmen momentan besitzt.

Es gibt allerdings auch Monitor-Programme, die die Farben nach einem BRK selber auf bestimmte Werte setzen. Das geht so schnell, daß Sie in einem solchen Fall kaum etwas von der Änderung sehen werden.

An diesem Programm können Sie zugleich das Editieren von Assembler-Programmen mit dem Monitor üben. Um dem Rahmen eine andere Farbe zu geben, ändern Sie den Wert $01 z.B. in $0A. Wenn sich die vom Monitor angezeigte Befehlszeile

```
A C000 A9 01            LDA #$01  (C64)
```

bzw.

```
A 03F7 A9 01            LDA #$01  (C16, Plus/4)
```

noch auf dem Bildschirm befindet, genügt es, den eigentlichen Assembler-Befehl (LDA #$01) zu überschreiben und RETURN zu drücken. Anschließend starten Sie das Programm erneut mit dem GO-Befehl.

Nicht nur STA, sondern auch LDA erlaubt den Einsatz der absoluten Adressierung. Anstelle eines Wertes kann dem LDA-Befehl eine absolute Speicheradresse angegeben werden (LDA $C000). In diesem Fall wird der Akkumulator mit dem Inhalt der angegebenen Speicherzelle geladen.

Die absolute Adressierung mit dem LDA-Befehls kann ebenfalls mit einem kleinen Assembler-Programm demonstriert werden.

```
Absolute Adressierung mit LDA (C64)

A C000 AD 0A C0         LDA $C00A
A C003 00               BRK

Absolute Adressierung mit LDA (C16, Plus/4)

A 03F7 AD FF 03         LDA $03FF
A 03FA 00               BRK
```

Verlassen Sie anschließend den Monitor und geben Sie POKE 49162,10 (C64) bzw. POKE 1023,10 (C16, Plus/4) ein. Rufen Sie den Monitor wieder auf und starten Sie das Programm mit SYS 49152 (C64) bzw. SYS 1015 (C16, Plus/4).

Die Registeranzeige des Monitors sollte nach dem BRK-Befehl für den Inhalt des Akkumulators den Wert $0A (=dezimal zehn) anzeigen. Um den Programmablauf zu klären, listen Sie bitte das Assembler-Programm mit dem Disassemble-Befehl ab Adresse $C000 (C64) bzw. ab $03F7 (C16, Plus/4).

Der Ablauf (C16, Plus/4 in Klammern):

1. Der POKE-Befehl schreibt in Speicherzelle 49162 = hexadezimal $C00A (1023 = $03FF) den Wert zehn, also die Hexadezimalzahl $0A, bevor das Assembler-Programm gestartet wird.

2. Der erste Assembler-Befehl LDA $C00A (LDA $03FF) kopiert den Inhalt der betreffenden Speicherzelle in den Akkumulator. Der Inhalt der Speicherzelle wird bei diesem Kopiervorgang nicht (!) verändert.

3. Der BRK-Befehl unterbricht das Assembler-Programm und ruft den Monitor auf, dessen Registeranzeige den Wert $0A als aktuellen Akkumulator-Inhalt ausgibt.

Der Akkumulator wurde im Demoprogramm mit Hilfe der absoluten Adressierung mit dem Inhalt der angegebenen Speicherzelle geladen (LDA (ADRESSE)), im Gegensatz zur erläuterten unmittelbaren Adressierung, bei der der Akkumulator mit einem unmittelbar angegebenen Wert geladen wird (LDA #(WERT)).

Am Programm-Listing erkennen Sie, daß die Länge eines Befehls von der Art der Adressierung abhängt. Der Befehl LDA #$01 besitzt eine Länge von zwei, der Befehl LDA $C00A eine Länge von drei Byte.

Der Grund: in beiden Fällen ist der eigentliche Befehlscode genau ein Byte lang. Das Argument ist jedoch bei der unmittelbaren Adressierung ein, bei der absoluten Adressierung zwei Byte lang.

Das Programm-Listing deckt einen weiteren interessanten Unterschied auf. Außer der Befehlslänge hängt auch der Befehlscode von der Art der Adressierung ab. Dem Befehl LDA #$01 entspricht die Bytefolge A9 01, dem Befehl LDA $C00A (LDA $03FF) die Bytefolge AD 0A C0 (AD FF 03).

Bei der unmittelbaren Adressierung wird der LDA-Befehl mit dem Code $A9 verschlüsselt, bei der absoluten Adressierung jedoch mit $AD.

Auch dafür ist ein einleuchtender Grund vorhanden. Wie erläutert, holt die CPU bei der Befehlsbearbeitung zuerst den Befehlscode und anschließend das Befehlsargument. Das Problem für die CPU besteht nun darin, anhand des Befehlscodes zu erkennen, ob ein Ein- oder ein Zwei-Byte-Argument folgt. Die Argumentlänge ist von der Adressierungsart abhängig. Daher werden für einen Befehl verschiedene Befehlscodes benutzt, je nachdem, welche Adressierungsart verwendet wird.

Am Code $A9 erkennt die CPU, daß es sich um einen LDA-Befehl mit unmittelbarer Adressierung handelt und daher ein Ein-Byte-Argument folgt, im Gegensatz zum Code $AD, der einen LDA-Befehl mit absoluter Adressierung kennzeichnet, dem ein Zwei-Byte-Argument folgt.

## 8.3. Die implizite Adressierung

Sie wissen bereits, daß die - bei den verschiedenen Commodore-Rechnern leicht unterschiedliche, jedoch vollkompatible - CPU verschiedene Register besitzt. Den Akkumulator lernten wir hinreichend kennen.

Außer dem Akkumulator stehen uns zwei sogenannte Indexregister zur Verfügung, das X-Register und das Y-Register. Diese Register werden vorwiegend zur Schleifenbildung, aber auch zur kurzfristigen Speicherung von Daten verwendet. Jedes dieser Register ist ebenso wie der Akkumulator ein Acht-Bit-Register und kann daher eine beliebige Zahl zwischen null und 255 aufnehmen, ein Byte.

Analog den Befehlen LDA und STA existieren Transferbefehle, mit denen Daten in diese Register transportiert oder umgekehrt die Inhalte der Register in eine anzugebende Speicherzelle kopiert werden können.

LDX (load X-Register, lade das X-Register) lädt das X-Register mit einem Wert. LDY (load Y-Register, lade das Y-Register) besitzt die gleiche Funktion für das Y-Register. Beide Befehle arbeiten analog zum LDA-Befehl, so daß sowohl die unmittelbare (LDX #$FF; LDY #$FF) als auch die absolute (LDX $C00A; LDY $C00A) Adressierung eingesetzt werden kann.

Entsprechend dem Befehl STA, mit dem der Akkumulator-Inhalt in eine beliebige Speicherzelle kopiert wird, stehen auch für die Indexregister X und Y entsprechende Befehle zur Verfügung.

Mit STX (store X-Register, übertrage das X-Register) und STY (store Y-Register, übertrage das Y-Register) wird der Inhalt eines Indexregisters in eine anzugebende Speicherstelle kopiert.

Das Demoprogramm zum Ändern der Rahmenfarbe kann daher jederzeit so umgeschrieben werden, das anstelle des Akkumulators eines der Indexregister verwendet wird.

```
Rahmenfarbe ändern (C64)

A C000 A2 01            LDX #$01
A C002 8E 20 0D         STX $D020
A C005 00               BRK
```

```
Rahmenfarbe ändern (C16, Plus/4)

A 03F7 A2 01          LDX #$01
A 03F9 8E 19 FF       STX $FF19
A 03FC 00             BRK
```

Der Befehl LDX #$01 lädt das X-Register mit dem unmittelbar (Kennzeichen "#") angegebenen Wert $01. Der Befehl STX $D020 (STX $FF19 für C16 und Plus/4) kopiert den Inhalt des X-Registers, d.h., den soeben in dieses Register geladenen Wert $01, in die absolut angegebene Speicherzelle $D020 ($FF19 für C16 und Plus/4).

Dieses Programm ist ebenso zu starten wie die vorigen Demoprogramme. Nach dem BRK-Befehl und der Rückkehr in das Monitor-Programm erkennen Sie an der Registeranzeige, daß das X-Register tatsächlich den Wert $01 enthält.

Es sollte Ihnen leichtfallen, die neu kennengelernten Befehle LDX, LDY, STX und STY zu handhaben, da sie sich kaum von den Befehlen LDA und STA unterscheiden.

Wichtig für uns ist jedoch, daß zusätzliche Befehle zum Datenaustausch zwischen (!) den verschiedenen Registern vorhanden sind. Der Inhalt des Akkumulators kann sowohl in das X- als auch in das Y-Register kopiert werden. Die Inhalte der X- und Y-Register lassen sich leider nicht direkt in das jeweils andere Indexregister übertragen.

Sowohl der Inhalt des X- als auch der Inhalt des Y-Registers kann jedoch in den Akkumulator kopiert werden.

Bei keiner dieser Transfer-Operationen geht der Inhalt des ursprünglichen Registers verloren!

Datenaustausch zwischen den Registern (Akku, X und Y):

- TAX (transfer accumulator to X-Register): der Inhalt des Akkumulators wird in das X-Register übertragen.
- TAY (transfer accumulator to Y-Register): der Inhalt des Akkumulators wird in das Y-Register übertragen.
- TYA (transfer Y-Register to accumulator): der Inhalt des Y-Registers wird in den Akkumulator übertragen.
- TXA (transfer X-Register to accumulator): der Inhalt des X-Registers wird in den Akkumulator übertragen.

Datenaustausch zwischen Arbeitsspeicher und CPU-Registern:

- LDA (load accumulator): der Akkumulator wird mit einem Wert geladen.
- LDX (load X-Register): das X-Register wird mit einem Wert geladen.
- LDY (load Y-Register): das Y-Register wird mit einem Wert geladen.
- STA (store accumulator): der Inhalt des Akkumulators wird in die angegebene Speicherzelle übertragen.
- STX (store X-Register): der Inhalt des X-Registers wird in die angegebene Speicherzelle übertragen.
- STY (store Y-Register): der Inhalt des Y-Registers wird in die angegebene Speicherzelle übertragen.

Die Transfer-Befehle zum Datenaustausch zwischen Registern verwenden nur eine Adressierungsart, die implizite Adressierung. Implizit bedeutet, daß das Argument im Befehl selbst bereits enthalten ist.

Für einen Befehl wie TXA wird keine weitere Angabe benötigt, da im Befehl selbst bereits enthalten ist, welcher Wert in den Akkumulator zu übertragen ist, eben der aktuelle Inhalt des X-Registers.

Bei dieser Befehlsklasse handelt es sich somit um Ein-Byte-Befehle, da dem Befehlscode kein Argument folgt.

Zur Übung schreiben wir ein Programm, das die ersten sechs Speicherzellen des Bildschirms abwechselnd mit den Zeichen A (Bildschirmcode eins) und B (Bildschirmcode zwei) beschreiben soll. Die herkömmliche Lösung:

```
Ausgabe der Zeichen A und B, Version 1 (C64)

A C000 A9 01          LDA #$01
A C002 8D 00 04       STA $0400
A C005 A9 02          LDA #$02
A C007 8D 01 04       STA $0401
A C00A A9 01          LDA #$01
A C00C 8D 02 04       STA $0402
A C00F A9 02          LDA #$02
A C011 8D 03 04       STA $0403
A C014 A9 01          LDA #$01
A C016 8D 04 04       STA $0404
A C019 A9 02          LDA #$02
A C01B 8D 05 04       STA $0405
A C01E 00             BRK

Ausgabe der Zeichen A und B, Version 1 (C16, Plus/4)

A 03F7 A9 01          LDA #$01
A 03F9 8D 00 0C       STA $0C00
A 03FC A9 02          LDA #$02
A 03FE 8D 01 0C       STA $0C01
A 0401 A9 01          LDA #$01
A 0403 8D 02 0C       STA $0C02
A 0406 A9 02          LDA #$02
A 0408 8D 03 0C       STA $0C03
A 040B A9 01          LDA #$01
A 040D 8D 04 0C       STA $0C04
A 0410 A9 02          LDA #$02
A 0412 8D 05 0C       STA $0C05
A 0415 00             BRK
```

Löschen Sie bitte den Bildschirm, bevor Sie das Programm wie gewohnt mit G C000 (G 03F7) starten, um ein eventuelles Scrollen zu vermeiden.

In der obersten Bildschirmzeile wird die Zeichenfolge ABABAB ausgegeben. Der Programmablauf ist folgendermaßen:

1. Der Akkumulator wird mit dem Wert $01 geladen, dem Bildschirmcode des Zeichens A. Der Inhalt des Akkumulators wird in die erste Speicherzelle des Bildschirmspeichers kopiert. Auf dem Bildschirm erscheint das Zeichen A.

2. Der Akkumulator wird mit dem Wert $02 geladen, dem Bildschirmcode des Zeichens B. Der Inhalt des Akkumulators wird in die zweite Speicherzelle des Bildschirmspeichers kopiert. Auf dem Bildschirm erscheint unmittelbar neben dem A das Zeichen B.

Die restlichen vier Zeichen werden auf die gleiche Weise ausgegeben, in den folgenden Speicherzellen des Bildschirmspeichers.

Der Ablauf dieses Programms bereitet Ihnen wohl kaum Schwierigkeiten. Wenn doch, lesen Sie bitte die Kapitel über die Befehle LDA, die unmittelbare und STA, die absolute Adressierung noch einmal sorgfältig durch.

Viel interessanter ist nun jedoch der Einsatz der neu erlernten Transferbefehle in diesem Programm. Das ständig wiederkehrende LDA #$01 und LDA #$02 kann mit diesen Befehlen erheblich vereinfacht werden.

Ausgabe der Zeichen A und B, Version 2 (C64)

```
A C000 A2 01          LDX #$01
A C002 A0 02          LDY #$02
A C004 8A             TXA
A C005 8D 00 04       STA $0400
A C008 98             TYA
A C009 8D 01 04       STA $0401
A C00C 8A             TXA
A C00D 8D 02 04       STA $0402
A C010 98             TYA
A C011 8D 03 04       STA $0403
A C014 8A             TXA
A C015 8D 04 04       STA $0404
A C018 98             TYA
A C019 8D 05 04       STA $0405
A C01C 00             BRK
```

Ausgabe der Zeichen A und B, Version 2 (C16, Plus/4)

```
A 03F7 A2 01          LDX #$01
A 03F9 A0 02          LDY #$02
A 03FB 8A             TXA
A 03FC 8D 00 0C       STA $0C00
A 03FF 98             TYA
A 0400 8D 01 0C       STA $0C01
A 0403 8A             TXA
A 0404 8D 02 0C       STA $0C02
A 0407 98             TYA
A 0408 8D 03 0C       STA $0C03
A 040B 8A             TXA
A 040C 8D 04 0C       STA $0C04
A 040F 98             TYA
A 0410 8D 05 0C       STA $0C05
A 0413 00             BRK
```

Der Programmablauf:

1. Das X- und das Y-Register werden am Programmanfang initialisiert. Die beiden Register erhalten die im weiteren Programmverlauf immer wieder benötigten Werte $01 (X-Register) und $02 (Y-Register).

2. TXA kopiert den Inhalt des X-Registers ($01) in den Akkumulator, der daraufhin den Wert $01 enthält. Der Inhalt des Akkumulators wird mit STA in Speicherzelle $0400 ($0C00) übertragen, das erste A wird ausgegeben.

3. TYA überträgt den Inhalt des Y-Registers ($02) in den Akkumulator. Der neue Akkumulatorinhalt $02 wird in Speicherzelle $0401 ($0C01) geschrieben, das erste B erscheint auf dem Bildschirm.

Der unter 2. und 3. geschilderte Ablauf wiederholt sich nun mehrmals. Beachten Sie unbedingt, daß Transferbefehle wie TXA und TYA den Inhalt der ursprünglichen Speicherzelle (Register) nicht verändern. Der Inhalt des X- und des Y-Registers bleibt während des gesamten Programmablaufs erhalten. Nur der Inhalt des Akkumulators wird durch diese Transferbefehle verändert.

Ein Vorteil dieses Programms besteht in der etwas geringeren Tipparbeit bei der Eingabe. Zudem ist das Programm ein wenig kürzer als die erste Version. Version 1 besitzt eine Länge von 31 Byte ($C000-$C01E bzw. $03F7-$0415 in der C16, Plus-Version), Version 2 eine Länge von 29 Byte ($C000-C01C bzw. $03F7-$0413).

Diese Unterschiede sind zweifellos gering. Bedenken Sie jedoch, daß das Programm zwei zusätzliche Befehle zur Initialisierung der Indexregister enthält, die bei dem relativ kurzen Programm ins Gewicht fallen, jedoch vernachlässigbar sind, wenn der gesamte Bildschirm mit A's und B's beschrieben wird.

In diesem Fall wird der Unterschied in der Programmlänge erheblich größer und es macht sich viel stärker bemerkbar, daß

der Zwei-Byte-Befehl LDA (WERT) durch den Ein-Byte-Befehl TXA bzw. TYA ersetzt wird.

Ein weiterer Unterschied betrifft die Ablaufgeschwindigkeit. Es ist unmittelbar einsichtig, daß der Prozessor gegenüber einem Befehl mit impliziter Adressierung wie TXA einen größeren Arbeitsaufwand erledigt, wenn er außer dem eigentlichen Befehlscode ein folgendes Argument aus dem Speicher lesen muß.

Vereinfacht ausgedrückt: Ein-Byte-Befehle werden schneller verarbeitet als Zwei-Byte-Befehle und diese wiederum schneller als Drei-Byte-Befehle.

Durch effektiven Einsatz der Indexregister können wir ein Programm daher erheblich beschleunigen. Zugegeben, die Ablaufgeschwindigkeit ist bei Assembler-Programmen weit weniger wichtig als bei Programmen, die in einer - erheblich langsameren - höheren Programmiersprache erstellt werden.

Dennoch werden auch Sie mit der Zeit wie jeder Assenmbler-Programmierer den Ehrgeiz entwickeln, Ihre Programme in Bezug auf Geschwindigkeit und Programmlänge so weit wie möglich zu optimieren.

## 8.4. Kurzadressierung (Zeropage-Adressierung)

Ich erwähnte bereits, daß die erste Seite des Rechnerspeichers, also der Bereich $0000-$00FF, eine besondere Rolle spielt. Ausschließlich mit dieser Zeropage ist die Kurzadressierung oder auch Zeropage-Adressierung möglich.

Die Besonderheit der Zeropage liegt darin, daß es möglich ist, jede ihrer Speicherzellen mit nur einem Byte zu adressieren ($00-$FF anstelle von $0000-$00FF).

Bei der Zeropage-Adressierung entfällt daher das High-Byte der Adresse. Ein Beispiel:

```
Zeropage-Adressierung (C64)

        A C000 A9 01            LDA #$01
        A C002 85 FE            STA $FE
        A C004 00               BRK

Zeropage-Adressierung (C16, Plus/4)

        A 03F7 A9 01            LDA #$01
        A 03F9 85 FE            STA $FE
        A 03FB 00               BRK
```

Es ist nicht unbedingt nötig, dieses Programm zu starten. Es schreibt den Wert $01 in die völlig unbedeutende Speicherzelle $00FE, ohne daß eine Wirkung erfolgt.

Anhand des Programm-Listings erkennen Sie jedoch die Vorteile der Zeropage-Adressierung. Anstelle der absoluten Adresse $00FE geben Sie die Kurzform $FE ein. Für die Adresse wird nur ein Byte benötigt. Der Befehl ist daher kürzer und wird schneller ausgeführt.

Aufgrund dieser Vorteile empfiehlt es sich, Daten, auf die häufig zugegriffen werden muß, möglichst in unbenutzen Speicherzellen der Zeropage abzulegen und bei den entsprechenden Transferbefehlen (LDA, STA, LDX, STX, LDY, STY) die Zeropage-Adressierung einzusetzen.

Unbenutzt und hervorragend zur Zeropage-Adressierung geeignete Bereiche der Zeropage sind vor allem:

```
C64: $FB-FE
C16, Plus/4: $D8-$E8
```

## 8.5. Indizierte Adressierung

Die indizierte Adressierung ist eine der schwierigsten, aber auch flexibelsten Adressierungarten, die uns zur Verfügung steht. Da sich mit dieser Adressierungsart viele Probleme äußerst elegant

lösen lassen, wird die indizierte Adressierung sehr häufig in Programmen eingesetzt.

Seit längerem verwenden wir bereits die Indexregister X und Y, ohne daß der Ausdruck Indexregister näher erläutert wurde. Der Ausdruck erklärt sich durch den Einsatz dieser Register bei der indizierten Adressierung.

Indiziert bedeutet, daß bei dieser Adressierungsart außer der eigentlichen Adresse auch ein Index eine Rolle spielt. Mit Indizes arbeiten Sie wahrscheinlich seit langem in Ihren BASIC-Programmen.

Üblicherweise genügt zum Zugriff auf eine bestimmte BASIC-Variable die Angabe des Variablennamens (PRINT A:PRINT F$...), mit einer Ausnahme: zum Zugriff auf eine Arrayvariable ist außer dem Namen des Arrays immer ein zusätzlicher Index anzugeben, wie im folgenden Beispiel:

```
100 FOR I=1 TO 10
110 : PRINT A$(I)
120 NEXT
```

In diesem Beispiel werden der Reihe nach die Inhalte der Arrayvariablen <A$(1)> bis <A$(10)> ausgegeben. Die Schleifenvariable <I> wird als Index benutzt.

Erst die Kombination aus dem Namen des Arrays und dem Index ergibt die komplette Adresse der Variablen.

Analog funktioniert die indizierte Adressierung. Die Adresse einer Speicherzelle ergibt sich aus einer angegebenen Adresse und einem zusätzlichen Index, genauer: aus der Adresse plus (!) dem Inhalt eines der Indexregister, wie im folgenden Beispiel:

```
A C000 A2 04        LDX #$04
A C002 BD 00 04     LDA $0400,X
A C005 00           BRK
```

In diesem Beispiel (das Sie nicht einzugeben brauchen, entscheidend ist das dahinterstehende Prinzip) wird zuerst das Indexregister X mit dem Wert $04 geladen (LDX #$04).

Der folgende Befehl LDA $0400,X zeigt den Einsatz der indizierten Adressierung. Die Adresse ergibt sich aus der absolut angegebenen Adresse $0400 plus (!) dem Inhalt des X-Registers ($04). Der Befehl LDA $0400,X lädt somit den Akkumulator mit dem Inhalt der Speicherzelle $0404.

Ein weiteres Beispiel:

```
A C000 A0 FF            LDY #$FF
A C002 B9 00 04         LDA $0400,Y
A C005 00               BRK
```

In diesem Beispiel wird das Y-Register mit $FF geladen. Mit dem Befehl LDA $0400,Y wird auf die Adresse $0400 plus dem Inhalt des Y-Registers zugegriffen, also auf die Speicherzelle $04FF.

Um zu sehen, daß die angesprochene Adresse auch wirklich vom Registerinhalt abhängt, geben Sie bitte zusätzlich folgendes Programm ein:

```
A C000 A9 01            LDA #$01
A C002 A0 00            LDY #$00
A C004 99 00 04         STA $0400,Y
A C007 00               BRK
```

Löschen Sie bitte den Bildschirm und starten Sie das Programm wie gewohnt mit G C000. In der linken oberen Ecke (erste Spalte der obersten Zeile) wird das Zeichen A ausgegeben.

Ändern Sie nun den Befehl LDY #$00, z.B. in LDY #$01, und starten Sie das Programm erneut. Das A wird nun in der zweiten Spalte der gleichen Zeile ausgegeben. Statt $0400 wird die Speicherzelle $0401 angesprochen.

Diese Adressierungsart stellt auf den ersten Blick nur eine umständlichere Version der Befehle STA $0400 bzw. STA $0401 dar, mit denen einfach direkt auf die gewünschten Speicherzellen zugegriffen werden kann.

Der Sinn der indizierten Adressierung liegt vorwiegend im Einsatz innerhalb von Programmschleifen, analog zum zuvor abgebildeten BASIC-Programm.

Diesem Ziel, der Programmierung von Schleifen, werden wir uns nun mit Riesenschritten nähern. Die folgenden Seiten liefern Ihnen das zur Schleifenbildung notwendige Wissen. Zusammen mit der indizierten Adressierung werden wir dann bereits in der Lage sein, höchst effektive Assembler-Programme zu erstellen.

Merken Sie sich bitte zur indizierten Adressierung:

1. Die Adresse ergibt sich aus der absolut angegebenen Adresse plus dem Inhalt des verwendeten Indexregisters.

2. Die implizite Adressierung kann nur den Inhalt des Akkumulators auf diese Weise übertragen. Ein Befehl wie STX $0400,X oder STX $0400,Y ist nicht zulässig!

## 9. Inkrementier- und Dekrementierbefehle

In Assembler-Programmen ist es oftmals notwendig, den Inhalt bestimmter Speicherzellen oder Register zu erhöhen bzw. zu vermindern.

Zu diesem Zweck stehen uns die sogenannten Inkrementier- (Inkrementieren=erhöhen) bzw. Dekrementier-Befehle (Dekrementieren=vermindern) zur Verfügung.

Der Befehl INC (ADRESSE) erhöht den Inhalt der angesprochenen Speicherzelle um eins. DEC (ADRESSE) vermindert den Inhalt der betreffenden Speicherzelle um eins.

Die Befehle lassen sich gut merken, wenn wir wissen, daß INC die Abkürzung für increment und DEC die Abkürzung für decrement ist.

Mit diesen Befehlen können wir sehr einfach ein Programm schreiben, mit dem wir uns eine Bildschschirmrahmenfarbe nach unserem Geschmack aussuchen können.

```
Rahmenfarbe ändern (C64)

A C000 EE 20 D0          INC $D020
A C003 00                BRK
A C004 CE 20 D0          DEC $D020
A C007 00                BRK

Rahmenfarbe ändern (C16, Plus/4)

A 03F7 EE 19 FF          INC $FF19
A 03FA 00                BRK
A 03FB CE 19 FF          DEC $FF19
A 03FE 00                BRK
```

Eigentlich handelt es sich hierbei um zwei Programme. Das erste Programm besteht aus dem Befehl INC $D020. $D020 (C16, Plus/4: $FF19) ist jene Speicherzelle, die die Farbe des Bildschirmrahmens in Form einer Zahl enthält. Wird diese Zahl verändert, ändert sich die Rahmenfarbe entsprechend.

Rufen Sie das Programm wie gewohnt mit G C000 (G 03F7) auf. Der Befehl INC $D020 erhöht den Inhalt dieser Speicherzelle um eins und die Rahmenfarbe ändert sich. Der BRK-Befehl unterbricht das Programm und führt zur Rückkehr in den Monitor. Jeder neue Aufruf des Programms ändert die Rahmenfarbe.

Dies funktioniert allerdings nur sichtbar, wenn der Monitor nicht anschließend beim BRK eigenständig die Farben zurücksetzt.

Auf diese Weise können Sie problemlos alle Farben ausprobieren. Sollten Sie über das Ziel hinausgeschossen sein und Ihnen die vorige Farbe besser gefallen, rufen Sie einfach den zweiten Programmteil mit G C004 (G 03FB) auf. Der Befehl DEC $D020 vermindert den Inhalt von $D020 wieder um eins.

Zusammen erlauben Ihnen die Programme, den Farbkatalog Ihres Rechners vor- und rückwärts zu durchblättern. Sollten Sie allerdings einen Monitor einsetzen, der seine eigenen Farben setzt, so funktioniert das Programm leider nicht.

Nicht nur der Inhalt von Speicherzellen, auch die Inhalte der Indexregister X und Y können inkrementiert/dekrementiert werden. Der Befehl INX (increment X) inkrementiert den Inhalt des X-Registers, und der Befehl INY (increment Y) inkrementiert das Y-Register. Entsprechend dekrementiert der Befehl DEX (decrement X) das X-Register und der Befehl DEY (decrement Y) das Y-Register.

Alle vier Befehle (INX, INY, DEX, DEY) verwenden die implizite Adressierung. Die Adresse des Registers, auf die sich der Befehl beziehen soll, ist bereits im Befehlswort selbst enthalten.

Entsprechende Befehle, zur Erhöhung/Verminderung des Akkumulators um eins, sind leider nicht vorhanden. Die gleiche Wirkung kann jedoch über einen Umweg erzielt werden. Der Inhalt des Akkumulators wird in eines der Indexregister übertragen (TAX oder TAY), das Indexregister in- oder dekrementiert (INX, INY bzw. DEX, DEY) und der neue Inhalt des betreffenden Registers in den Akkumulator zurückgeschrieben (TXA oder TYA):

```
Akkumulator inkrementieren     Akkumulator dekrementieren

 TAX              TAY           TAX              TAY
 INX      oder    INY           DEX      oder    DEY
 TXA              TYA           TXA              TYA
```

*Inkrementier-/Dekrementierbefehle*

- INC (ADRESSE): Erhöht den Inhalt der angesprochenen Speicherzelle um eins.
- DEC (ADRESSE): Vermindert den Inhalt der angesprochenen Speicherzelle um eins.
- INX: Erhöht den Inhalt des X-Registers um eins.
- DEX: Vermindert den Inhalt des X-Registers um eins.
- INY: Erhöht den Inhalt des Y-Registers um eins.
- DEY: Vermindert den Inhalt des Y-Registers um eins.

Um den Umgang mit dieser Vielzahl neuer - aber sehr ähnlicher - Befehle zu erlernen, sollten Sie mehrere kleine Programme damit erstellen.

## 10. Programmschleifen

Der folgende Abschnitt führt Sie in die Schleifenbildung ein. Ebenso wie in BASIC sind auch in Assembler größere Programme ohne den Einsatz von Schleifen undenkbar.

Assembler unterscheidet sich von BASIC jedoch unter anderem durch die enorme Vielfalt verschiedener Schleifenbefehle, die die unterschiedlichsten Schleifenkonstruktionen ermöglicht.

Im folgenden wird anhand eines Schleifenbefehls - eines Branch-Befehls - die Grundform einer Assembler-Schleife demonstriert.

### 10.1. Schleifen bilden mit BNE

Sie ahnen bestimmt, daß es mit den In- und Dekrementierbefehlen möglich ist, Schleifen zu bilden. In BASIC werden Schleifen ja ebenfalls gebildet, indem eine Zählvariable erhöht oder vermindert wird. Der Schleifenzähler wird in Assembler-Programmen meist nach jedem Durchgang um eins vermindert, wie auch im folgenden BASIC-Programm:

```
100 I=100:REM STARTWERT FUER SCHLEIFENZAEHLER
110 POKE 53280,I:REM SCHLEIFENBEFEHLE
120 I=I-1:REM SCHLEIFENZAEHLER VERMINDERN
130 IF I<>0 THEN GOTO 110:REM BEREITS 100 DURCHGAENGE?
```

Analog zu diesem BASIC-Programm werden auch in Assembler Schleifen gebildet. Eine Zählvariable erhält einen Startwert, die Schleifenbefehle werden bearbeitet und die Zählvariable am Schleifenende vermindert.

Diesen Vorgang können wir problemlos mit den kennengelernten Befehlen nachbilden. Als Zählvariable verwenden wir eines der zwei Indexregister X und Y, da der Inhalt dieser Register mit einem kurzen Ein-Byte-Befehl problemlos erhöht oder vermindert werden kann.

```
A C000 A2 64      LDX #$64
A C002 8E 20 D0   STX $D020
A C005 CA         DEX
```

Der Befehl LDX #$64 lädt das X-Register - unseren Schleifenzähler - mit dem Wert 100 ($64), analog zum BASIC-Befehl I=100. Der Befehl STX $D020 entspricht dem Befehl POKE 53280,I und überträgt den Inhalt des X-Registers in $D020, jene Speicherzelle, die die Rahmenfarbe bestimmt. DEX vermindert den Inhalt unserer Zählvariablen ebenso wie I=I-1 um eins.

Den BASIC-Befehl IF I<>0 THEN GOTO 110 können wir mit dem Erlernten noch nicht umsetzen. Uns fehlt ein Vergleichsbefehl, mit dem wir in Abhängigkeit von einer Bedingung zum Schleifenanfang zurückkehren können.

Der benötigte Befehl lautet BNE (ADRESSE). Es handelt sich um einen sogenannten Branch- oder auch Verzweigungs-Befehl. BNE entspricht der Kombination der BASIC-Befehle IF und GOTO. Die auf IF folgende Bedingung ist allerdings festgelegt. Verzweigt wird immer dann, wenn die Zählvariable ungleich null ist.

Zur Erläuterung muß ich kurz auf das sogenannte Status-Register des Prozessors eingehen. Ich erwähnte bereits, daß uns dieses Statusregister Auskunft über verschiedene Zustände des Rechners gibt.

Das Status-Register enthält wie üblich ein Byte, also acht Bits. Jedes dieser Bits besitzt eine spezielle Bedeutung. Uns interessiert momentan ausschließlich Bit 1, das sogenannte Zero-Flag. Immer dann, wenn ein Befehl, der eine Speicherzelle oder ein Register verändert (INY, DEX, DEC...), das Ergebnis null zur Folge hat, wird dieses Bit gesetzt und erhält den Wert eins.

Beispiel: Wenn das X-Register den Inhalt $01 besitzt, führt ein nachfolgender DEX-Befehl zum Ergebnis null im X-Register. Automatisch wird nun im Status-Register das Zero-Flag gesetzt, das Bit 1 des Status-Registers enthält die Ziffer eins.

Im Gegensatz dazu wird dieses Flag gelöscht (enthält die Ziffer null), wenn die vorhergehende Operation zu irgendeinem Ergebnis ungleich null führte.

Beispiel: Wenn das X-Register den Inhalt eins besitzt, führt ein folgender INX-Befehl zum Ergebnis zwei (X enthält nun den Wert zwei). Da das Ergebnis des DEX-Befehls zwei ist, also ein Wert ungleich null, wird das Zero-Flag gelöscht.

Wichtig für uns ist an diesem Vorgang, daß wir uns nicht im geringsten um dieses Setzen oder Löschen des Zero-Flags kümmern müssen. Diese Aufgabe übernimmt der Rechner für uns.

Der BNE-Befehl überprüft, ob das Zero-Flag momentan gesetzt oder gelöscht ist. Wenn es gesetzt ist - also der vorausgegangene DEX-Befehl oder eine beliebige andere Operation zum Ergebnis null führte - erfolgt ein Sprung zur angegebenen Adresse. Die Arbeitsweise dieses Befehls verdeutlicht der volle Befehlsname. BNE bedeutet branch if not equal zero, verzweige, wenn nicht gleich null.

Mit dem BNE-Befehl können wir unser Programm vervollständigen:

```
Assembler                                BASIC

A C000 A2 64          LDX #$64           100 I=100
A C002 8E 20 D0       STX $D020          110 POKE 53280,I
A C005 CA             DEX                120 I=I-1
A C006 D0 FA          BNE $C002          130 IF I<>0 THEN GOTO 20
A C008 00             BRK
```

Dieses Programm entspricht nun exakt dem BASIC-Programm zur Änderung der Rahmenfarbe. Wenn Sie es auf Ihrem C64 starten, werden Sie allerdings von dieser Veränderung sehr wenig mitbekommen. Das Programm schreibt die Werte 100,99,..,1,0 einfach zu schnell in die Speicherzelle $D020. Die Schleife läuft für das menschliche Auge mit viel zu hoher Geschwindigkeit ab.

Der Programmablauf:

1. Das X-Register wird durch LDX #$64 mit dem Wert $64 (=dezimal 100) geladen.

2. Dieser Wert wird in die Speicherzelle $D020 kopiert (STX $D020) und dadurch die Rahmenfarbe entsprechend geändert.

3. Der Befehl DEX vermindert den Inhalt des X-Registers um eins, das nun den Wert $63 (=dezimal 99) enthält. Beachten Sie bitte, daß diese Operation ein Ergebnis besitzt, das ungleich null ist, nämlich 99, und das Zero-Flag gelöscht wird.

4. Der Befehl BNE $C002 überprüft anhand des Zero-Flags, ob die letzte Operation das Ergebnis null erbrachte, was an einem gesetzten Zero-Flag erkannt wird. Da dies nicht der Fall war (das Zero-Flag ist gelöscht), wird der Sprung zu Adresse $C002 ausgeführt.

Beachten Sie bitte, daß es für die praktische Benutzung dieses Befehls völlig uninteressant ist, wie (!) das Zero-Flag überprüft wird. Uns interessiert nur, daß das Zero-Flag gesetzt wird, wenn der Befehl DEX zum Ergebnis null führt, und der folgende BNE-Befehl dieses Ergebnis anhand des Zero-Flags überprüfen kann.

Solange der DEX-Befehl nicht zum Ergebnis null führt, ist die Bedingung "verzweige, wenn nicht gleich null" erfüllt und der nächste Schleifendurchgang beginnt. Die Frage ist nun, wann die Schleife beendet ist.

Bei jedem Schleifendurchgang wird der Inhalt des X-Registers mit dem Befehl DEX um eins vermindert. Nach 99 Durchgängen enthält das X-Register somit den Wert eins. Nun folgt wiederum der DEX-Befehl. Das X-Register wird dekrementiert und erhält den Wert null.

Der DEX-Befehl führte somit zum Ergebnis null und die Bedingung "verzweige, wenn nicht gleich null" ist nicht erfüllt. Diesmal verzweigt der BNE-Befehl nicht zum Befehl an Adresse $C002, zum Schleifenanfang.

Das Programm wird wie in BASIC mit jenem Befehl fortgesetzt, der dem BNE-Befehl folgt (BRK), die Schleife ist nach dem hundersten Durchlauf beendet.

Da Schleifen von ungeheuerer Wichtigkeit bei der Programmierung sind, fasse ich das gesagte kurz zusammen:

1. Um Schleifen zu bilden, laden wir eines der Indexregister mit einem Startwert, der der gewünschten Anzahl von Schleifendurchgängen entspricht.

2. Wie von BASIC gewohnt, folgen die Befehle, die innerhalb der Schleife bearbeitet werden sollen.

3. Am Schleifenende dekrementieren wir unser Zählregister (X- oder Y-Register).

4. Es folgt ein BNE-Befehl mit der Adresse des ersten Schleifenbefehls. Solange der vorhergehende Befehl, die Dekrementierung des Indexregisters, nicht den Wert null ergibt, verzweigt das Programm zur angegebenen Adresse.

## 10.2. Verzögerungsschleifen

Da wir im folgenden sehr häufig mit Schleifen arbeiten werden, ist es angesichts der Geschwindigkeit von Assembler angebracht, für eine gewisse Verzögerung zu sorgen, um den Ablauf der Demoprogramme verfolgen zu können.

Sie wissen bereits von BASIC-Programmen, was unter einer Verzögerungsschleife zu verstehen ist: eine Schleife, die keinerlei Funktion außer der Verlangsamung des Programmablaufs besitzt. Eine solche Schleife besteht aus Schleifenanfang und Schleifenende ohne die sonst üblichen Befehle innerhalb der Schleife. In BASIC:

```
100 FOR I=1 TO 1000
110 NEXT I
```

Auch in Assembler sind derartige Verzögerungsschleifen möglich. Die maximale Anzahl an Schleifendurchgängen beträgt jedoch 255, wie im folgenden Programm:

```
A C000 A2 FF        LDX #$FF
A C002 CA           DEX
A C003 DO FD        BNE $C002
```

Assembler ist jedoch derartig schnell, daß selbst 255 Durchgänge für eine Verzögerungsschleife noch viel zu wenig sind. Die vorgestellte Schleife wird in kaum feststellbarer Zeit abgearbeitet.

Um die Verzögerungszeit zu verlängern, wäre nun die Schachtelung zweier Schleifen möglich, analog zum folgenden BASIC-Programm.

```
100 FOR I=1 TO 255
110 : FOR J=1 TO 255
120 : NEXT J
130 NEXT I
```

Die innere Schleife wird in diesem Programm 255mal durchlaufen, bevor ein weiterer Durchgang der äußeren Schleife beginnt, die ebenfalls 255mal durchlaufen wird. Es ergeben sich somit 255*255=65025 Schleifendurchgänge. Das entsprechende Assembler-Programm:

Verzögerungsschleifen (C64)

```
A C000 A2 FF            LDX #$FF
A C002 A0 FF            LDY #$FF
A C004 88               DEY
A C005 D0 FD            BNE $C004
A C007 CA               DEX
A C008 D0 F8            BNE $C002
A C00A 00               BRK
```

Verzögerungsschleifen (C16, Plus/4)

```
A 03F7 A2 FF            LDX #$FF
A 03F9 A0 FF            LDY #$FF
A 03FB 88               DEY
A 03FC D0 FD            BNE $03FB
A 03FE CA               DEX
A 03FF D0 F8            BNE $03F9
A 0401 00               BRK
```

Wichtig für das Verständnis dieses Programms ist das Erkennen der Schleifenführung. Die innere Schleife besteht aus den Befehlen:

```
          LDY #$FF
(ADRESSE) DEY
          BNE $(ADRESSE)
```

Das Y-Register wird mit dem Wert $FF initialisiert. Der folgende Befehl DEY dekrementiert Y, das anschließend den Wert $FE enthält. Da das Ergebnis von DEY ungleich null ist, wird der bedingte Sprung BNE ausgeführt und der nächste Schleifendurchgang beginnt. Nach 255 Durchgängen führt der DEY-Befehl zum neuen Inhalt null des Y-Registers. Aufgrund dieses Ergebnisses null ist die Bedingung BNE ("verzweige, wenn ungleich null") nicht erfüllt und die innere Schleife wird verlassen.

Der folgende Befehl DEX dekrementiert den Schleifenzähler der äußeren Schleife, das X-Register. Solange dieser DEX-Befehl nicht den Wert null ergibt, wird der bedingte Sprung BNE zum Beginn der äußeren Schleife ausgeführt. Das Y-Register wird erneut mit dem Startwert $FF geladen und die innere Schleife erneut 255mal durchlaufen.

Nach dem 255ten Durchlauf der äußeren Schleife führt der Befehl DEX zum Resultat null als neuem Inhalt des X-Registers. Da der folgende bedingte Sprung somit nicht mehr ausgeführt wird, wird das Programm mit dem ersten Befehl fortgesetzt, der der Verzögerungsroutine folgt.

Diese geschachtelten Verzögerungsschleifen bewirken eine Verzögerung von knapp einer Sekunde, die für unsere Zwecke völlig ausreicht.

Anhand dieses Programms will ich Ihnen jedoch zum ersten Mal einen Vorgeschmack auf die Betriebssystem-Routinen geben. Das Betriebssystem stellt uns eine Unmenge nützlicher Unterprogramme (Routinen) zur Verfügung, unter anderem eine Verzögerungsroutine.

Die Verzögerungszeit dieser Routine beträgt eine Millisekunde (C64). Wenn wir den Aufruf dieser Routine in einer Schleife 255mal vornehmen, beträgt die Gesamtverzögerung somit 255 Millisekunden oder eine viertel Sekunde.

Das Prinzip:

```
          LDX #$FF
(ADRESSE) JSR $(VERZÖGERUNG)
          DEX
          BNE (ADRESSE)
```

Der Aufruf der Verzögerungsroutine erfolgt mit dem Befehl JSR $EEB3 ($E2DC beim C16, Plus/4, wobei die Verzögerungszeit etwas kürzer ist als beim C64), der dem BASIC-Befehl GOSUB entspricht.

Nach jedem Aufruf wird das zu Beginn mit $FF geladene X-Register dekrementiert. Nach dem 255ten Durchlauf führt die Dekrementierung zum Ergebnis null und die Schleife wird verlassen, der bedingte Sprung nicht mehr ausgeführt.

Verzögerung mit Betriebssystem-Routine (C64)

```
A C000 A2 FF          LDX #$FF
A C002 29 B3 EE       JSR $EEB3
A C005 CA             DEX
A C006 DO FA          BNE $C002
A C008 00             BRK
```

Verzögerung mit Betriebssystem-Routine (C16, Plus/4)

```
A 03F7 A2 FF          LDX #$FF
A 03F9 29 DC E2       JSR $E2DC
A 03FC CA             DEX
A 03FD DO FA          BNE $03F9
A 03FF 00             BRK
```

Beim Einbau dieser Verzögerungsroutine in unsere Programme ist zu beachten, daß durch die Schleife der ursprüngliche Wert des X-Registers natürlich verändert wird. Wenn die Schleife verlassen wird, besitzt das X-Register den Wert null, unabhängig vom Wert, den es vor dem Eintritt in die Schleife besaß.

Die Verzögerungsroutine wird an der gewünschten Position in ein Programm eingefügt. Soll der ursprüngliche Inhalt des X-Registers auch nach dem Durchlaufen der Schleife erhalten bleiben, verwenden wir am besten das Y-Register als Zählvariable.

Beim Einbau des Programms muß weiterhin beachtet werden, daß auch der Inhalt des Akkumulators verändert wird, und zwar durch die aufgerufene Betriebssystem-Routine (obwohl dies anhand des Aufrufs nicht unmittelbar zu erkennen ist).

Um die Routine praktisch zu erproben, werden wir sie in das Programm zur Änderung der Bildschirmfarben einbauen. Da in diesem Programm (siehe oben) der Inhalt des X-Registers durch die Verzögerungsschleife nicht willkürlich verändert werden darf, verwenden wir das Y-Register als Schleifenzähler.

```
Bildschirmfarben verzögert ändern (C64)

A C000 A2 64          LDX #$64
A C002 8E 20 D0       STX $D020
A C005 A0 FF          LDY #$FF
A C007 20 B3 EE       JSR $EEB3
A C00A 88             DEY
A C00B D0 FA          BNE $C007
A C00D CA             DEX
A C00E D0 F2          BNE $C002
A C010 00             BRK
```

```
Bildschirmfarben verzögert ändern (C16, Plus/4)

A 03F7 A2 64          LDX #$64
A 03F9 8E 20 D0       STX $FF19
A 03FC A0 FF          LDY #$FF
A 03FE 20 B3 EE       JSR $E2DC
A 0401 88             DEY
A 0402 D0 FA          BNE $03FE
A 0404 CA             DEX
A 0405 D0 F2          BNE $03F9
A 0407 00             BRK
```

## 10.3. Effektiver Einsatz von Schleifen

Bei der Erläuterung der indizierten Adressierung versprach ich Ihnen, daß diese zusammen mit Programmschleifen bereits zu recht wirkungsvollen Ergebnissen führt.

Um diese Behauptung zu beweisen, greife ich auf eines unserer Ausgangsbeispiele zurück, das Füllen des Bildschirms mit dem Zeichen A.

Zu Beginn unseres Kurses war dies noch eine recht mühsame Angelegenheit und erforderte eine Unmenge einzelner STA-Befehle.

```
A C000 A9 01          LDA #$01
A C002 8D 00 04       STA $0400
A C005 8D 01 04       STA $0401
A C008 8D 02 04       STA $0402
...
...
```

Ein solches Programm wird bei Ihnen sicher keine Begeisterungsausbrüche auslösen. Eine umständlichere Art und Weise zum Beschreiben des Bildschirms ist kaum denkbar.

Das Problem kann jedoch unter Einsatz der indizierten Adressierung und einer Schleife weit einfacher gelöst werden:

```
Zeichenausgabe mit einer Schleife (C64)

A C000 A9 01          LDA #$01
A C002 A2 FF          LDX #$FF
A C004 9D 00 04       STA $0400,X
A C007 CA             DEX
A C008 D0 FA          BNE $C004
A C00A 00             BRK
```

Zeichenausgabe mit einer Schleife (C16, Plus/4)

```
A 03F7 A9 01          LDA #$01
A 03F9 A2 FF          LDX #$FF
A 03FB 9D 00 0C       STA $0C00,X
A 03FE CA             DEX
A 03FF D0 FA          BNE $03FB
A 0401 00             BRK
```

Sie erkennen sicherlich die von $C004-$C008 ($03FB-$03FF) reichende Schleife. Wie im letzten Kapitel wird die Schleife gebildet, indem das X-Register mit der gewünschten Anzahl der Schleifendurchgänge geladen und in jedem Schleifendurchgang dekrementiert wird.

Die Schleife wird ausgeführt, solange der DEX-Befehl nicht zum Ergebnis null führt und dadurch das Zero-Flag gesetzt wird. Solange dieser Fall nicht eintritt, verzweigt der BNE-Befehl immer zum Schleifenanfang.

Der Befehl STA $0400,X (STA $0C00,X) arbeitet mit indizierter Adressierung. Die Endadresse ergibt sich aus der angegebenen Adresse plus dem aktuellen Inhalt des X-Registers. Beim ersten Schleifendurchgang besitzt X noch den Startwert $FF. Die Adresse ergibt sich somit als $04FF ($0CFF).

In diese aus angegebener Adresse plus X-Register resultierende Adresse wird der Inhalt des Akkumulators geschrieben. Da der Bildschirmspeicher an Adresse $0400 ($0C00) beginnt, wird in die 255te Bildschirmspeicherzelle der Bildschirmcode für A geschrieben.

Beim zweiten Durchgang ergibt sich als resultierende Adresse $04FE ($0CFE), da das X-Register um eins vermindert wurde und nun den Wert $FE besitzt ($0400 + $FE = $04FE).

Das Zeichen A wird somit in die 254te Speicherzelle des Video-RAM übertragen. Sie erkennen das allgemeine Schema: nach jedem Schleifendurchlauf ist der Wert des X-Registers um eins

niedriger als zuvor und daher auch die Endadresse ($0400 + X-Register).

Im letzten Schleifendurchgang enthält das X-Register den Wert eins. In Speicherzelle $0401 ($0C01) wird der Code für A geschrieben. Anschließend wird das X-Register dekrementiert und es folgt wieder der bedingte Sprungbefehl BNE, der in diesem Fall nicht ausgeführt wird, da das Ergebnis der letzten Operation (DEX) zum Ergebnis null führte.

Das Programm füllt somit der Reihe nach die Speicherzellen $04FF-$0401 ($0CFF-$0C01) mit dem Bildschirmcode des Zeichens A.

Das Programm entspricht bis in Details folgendem BASIC-Programm:

```
100 A=1:REM BILDSCHIRMCODE FUER A
110 X=255:REM ZAEHLER INITIALISIEREN
120 POKE 1024+I,A:REM A IN VIDEO-RAM
130 X=X-1:REM ZAEHLER DEKREMENTIEREN
140 IF X<>0 THEN GOTO 120:REM BEDINGTER SPRUNG
```

Das BASIC-Programm ist auch auf dem C16, Plus/4 lauffähig, wenn Sie die Startadresse des Bildschirmspeichers 1024 ($0400) durch 3072 ($0C00) ersetzen.

Vergleichen Sie Assembler- und BASIC-Programm und probieren Sie beide aus. Der Geschwindigkeitsunterschied ist geradezu unglaublich. Das Assembler-Programm ist übrigens nur elf Byte lang, im Gegensatz zu dem knapp 50 (!) Byte langen BASIC-Programm, das zudem weiteren Speicherplatz für die verwendeten Variablen benötigt.

Da wir inzwischen bereits bei reichlich komplexen Programmen angelangt sind, stelle ich ein weiteres Demoprogramm vor. Wie wäre es denn zur Abwechslung (um die immer wiederkehrende sinnlose Ausgabe von A's zu vermeiden), wenn wir uns den Zeichensatz unseres Rechners ausgeben lassen?

Die Ausgabe des kompletten Zeichensatzes ist mit einem winzigen Programm möglich, das nicht umfangreicher als das vorige ist:

```
Zeichensatz-Ausgabe (C64)

A C000 A2 FF            LDX #$FF
A C002 8A               TXA
A C003 9D 00 04         STA $0400,X
A C006 CA               DEX
A C007 D0 F9            BNE $C002
A C009 00               BRK

Zeichensatz-Ausgabe (C16, Plus/4)

A 03F7 A2 FF            LDX #$FF
A 03F9 8A               TXA
A 03FA 9D 00 0C         STA $0C00,X
A 03FD CA               DEX
A 03FE D0 F9            BNE $03F9
A 0400 00               BRK
```

Das X-Register wird mit dem Wert $FF initialisiert. Im folgenden Schleifendurchgang wird dieser Wert in den Akkumulator und von dort in die Bildschirmzelle $04FF ($CFF) übertragen. Diese Adresse ergibt sich aus der bei der indizierten Adressierung angegebenen Ausgangsadresse plus dem Inhalt $FF des X-Registers.

Das X-Register wird dekrementiert und - da das Ergebnis ungleich null ist (X=$FE) - der Branch-Befehl BNE ausgeführt.

Im folgenden Durchgang wird der Inhalt des X-Registers ($FE) wieder in den Akkumulator übertragen und in die Speicherzelle $04FE ($0CFE) geschrieben.

Der Reihe nach werden somit die Speicherzellen $04FF-$0401 ($0CFF-$0C01) mit den Bildschirmcodes $FF-$01 gefüllt. Die Schleife läuft wie üblich rückwärts, der Schleifenzähler wird nach jedem Durchgang um eins vermindert. Das Zeichen mit

dem Code $00 wird leider nicht mehr behandelt, da die Schleife verlassen wird, wenn die Dekrementierung des X-Registers zum Ergebnis null führt.

Mit dem bisher Erlernten können wir dieses Problem bereits lösen, indem wir nach dem Verlassen der Schleife das Zeichen mit dem Code $00 per Hand in die Speicherzelle $0400 ($0C00) schreiben und Sie den folgenden Befehl hinter dem Branch-Befehl einfügen:

```
...
...
BNE...
STX $0400 ($0C00 beim C16, Plus/4)
BRK
```

Im ersten Moment kommt Ihnen der Befehl STX $0400 wahrscheinlich sonderbar vor. Anbieten würde sich die Befehlsfolge:

```
LDA #$00
STA $0400
```

Bedenken Sie jedoch, daß die Schleife genau dann verlassen wird, wenn das X-Register den Wert null annimmt. In diesem Moment wird der bedingte Sprung BNE nicht mehr ausgeführt und der erste Befehl nach der Schleife ausgeführt.

Wir wissen daher mit absoluter Sicherheit, daß das X-Register nach Verlassen der Schleife den Wert null enthält. Ein Laden des Akkumulators oder eines Indexregisters mit $00 ist daher überflüssig. Wir können die geschilderte Tatsache ausnutzen und den Inhalt des X-Registers direkt in die Speicherzelle $0400 schreiben.

Bevor ich diesen Schnellkurs der Schleifenprogrammierung abschließe, empfehle ich Ihnen dringend, das Erlernte anhand kleiner Programmieraufgaben zu vertiefen.

Um Ihre Phantasie anzuregen, schließe ich diesen Abschnitt mit einem Programm ab, das einen Ball über den Bildschirm bewegt.

Das Programm arbeitet mit den erläuterten Verzögerungsschleifen. Die Geschwindigkeit der Bewegung können Sie beliebig variieren, indem Sie den Startwert des Schleifenzählers entsprechend ändern.

Versuchen Sie den Programmblauf bitte selbst zu durchschauen. Als Hilfestellung befinden sich Kommentare (die Sie bitte nicht abtippen!) hinter den einzelnen Befehlen.

Beachten Sie bitte, daß der Ball an einer bestimmten Bildschirmposition zuerst ausgegeben und nach Ablauf der Verzögerungsschleife wieder gelöscht - mit dem Leerzeichen überschrieben - wird (sonst ist der Bildschirm schnell mit Bällen gefüllt).

Starten Sie das Programm wie üblich mit G C000 (C64) bzw. G 03F7 (C16, Plus/4).

```
Ball bewegen (C64)

A C000 A2 FF      LDX #$FF      ;STARTWERT SCHLEIFENZAEHLER
A C002 A9 57      LDA #$57      ;BILDSCHIRMCODE FUER 'BALL'
A C004 9D 00 04   STA $0400,X   ;'BALL' AN $0400+X AUSGEBEN
A C007 A0 FF      LDY #$FF      ;STARTWERT VERZOEGERUNGSSCHLEIFE
A C009 20 B3 EE   JSR $EEB3     ;AUFRUF VERZOEGERUNGS-ROUTINE
A C00C 88         DEY           ;Y-REG.DEKREMENTIEREN
A C00D D0 FA      BNE $C009     ;Y=0? WENN NEIN, SPRUNG ZU $C009
A C00F A9 20      LDA #$20      ;BILDSCHIRMCODE F.LEERZEICHEN
A C011 9D 00 04   STA $0400,X   ;BALL LOESCHEN AN $0400+X
A C014 CA         DEX           ;SCHLEIFENZAEHLER DEKREMENTIEREN
A C015 D0 EB      BNE $C002     ;X=0? WENN NEIN, SPRUNG ZU $C002
A C017 00         BRK           ;UNTERBRECHUNG => MONITOR
```

```
Ball bewegen (C16, Plus/4)

A 03F7 A2 FF       LDX #$FF      ;STARTWERT SCHLEIFENZAEHLER
A 03F9 A9 57       LDA #$57      ;BILDSCHIRMCODE FUER 'BALL'
A 03FB 9D 00 0C    STA $0C00,X   ;'BALL' AN $0C00+X AUSGEBEN
A 03FE A0 FF       LDY #$FF      ;STARTWERT VERZOEGERUNGSSCHLEIFE
A 0400 20 DC E2    JSR $E2DC     ;AUFRUF VERZOEGERUNGS-ROUTINE
A 0403 88          DEY           ;Y-REG.DEKREMENTIEREN
A 0404 D0 FA       BNE $0400     ;Y=0? WENN NEIN, SPRUNG ZU $0400
A 0406 A9 20       LDA #$20      ;BILDSCHIRMCODE F.LEERZEICHEN
A 0408 9D 00 0C    STA $0C00,X   ;BALL LOESCHEN AN $0C00+X
A 040B CA          DEX           ;SCHLEIFENZAEHLER DEKREMENTIEREN
A 040C D0 EB       BNE $03F9     ;X=0? WENN NEIN, SPRUNG ZU $03F9
A 040E 00          BRK           ;UNTERBRECHUNG => MONITOR
```

## 11. Die wichtigsten Routinen des Betriebssystems

Das Betriebssystem des C64, C16, Plus/4 und des C128 enthält verschiedene Programme, die uns das Programmiererleben erheblich erleichtern. Alle Routinen sind Unterprogramme, vergleichbar mit einem BASIC-Unterprogramm.

BASIC-Unterprogramme werden mit GOSUB aufgerufen und mit RETURN beendet, wonach die Rückkehr zum aufrufenden Programm erfolgt. Assembler-Unterprogramme enden mit dem Befehl RTS (return from subroutine, kehre vom Unterprogramm zurück), der dem BASIC-RETURN entspricht.

Aufgerufen werden diese Routinen mit dem Befehl JSR (jump to subroutine, verzweige zu einem Unterprogramm), dem die Startadresse der Routine folgt. Die Adressen der in diesem Buch verwendeten Betriebssystem-Routinen sind - bis auf die bereits benutzte Verzögerungs-Routine - bei allen Commodore-Rechnern identisch.

Aus diesem Grund sind die vorgestellten Demoprogramme fast unverändert sowohl auf dem C64 als auch auf C16 und Plus/4 lauffähig und werden nur in der C64-Version vorgestellt.

Für C16, Plus/4-Besitzer beschränken sich die Änderungen in der Regel auf die Eingabe des Programms mit der Startadresse $03F7 anstatt $C000. Eine Ausnahme bilden Sprungbefehle. Sowohl beim bereits erläuterten Befehl BNE als auch den folgenden Befehlen BEQ und JMP ändern Sie bitte für den C16, Plus/4 die Sprungadressen entsprechend den vorangegangenen Demoprogrammen.

Bei der Benutzung dieser Routinen ist die sogenannte Parameterübergabe wichtig. Den Routinen werden verschiedene Werte übergeben (meist im Akkumulator und den Indexregistern), die die genaue Funktionsweise bestimmen.

In der Praxis ist weiter zu beachten, welche Register und Speicherzellen durch das Unterprogramm verändert werden. Sonst passiert es uns, daß wir in einer Schleife das X-Register als Schleifenzähler verwenden, der Inhalt dieses Registers jedoch durch das in der Schleife aufgerufene Unterprogramm verändert wird.

Im folgenden stelle ich jene Routinen vor, die am einfachsten zu benutzen sind und dennoch am häufigsten benötigt werden. Außer der Funktion und der Startadresse wird beschrieben, welche Register die betreffenden Routinen beeinflussen.

### 11.1. Zeichenausgabe mit BSOUT

Mit der Routine BSOUT (Adresse $FFD2) kann ein beliebiges Zeichen auf dem Bildschirm ausgegeben werden. Der ASCII-Code (nicht der Bildschirmcode!) des gewünschten Zeichens wird der Routine im Akkumulator übergeben, bevor der Aufruf mit JSR $FFD2 erfolgt.

Das Zeichen wird an jener Bildschirmposition ausgegeben, an der sich momentan der Cursor befindet und dieser eine Spalte weiterbewegt (BSOUT arbeitet analog dem PRINT-Befehl in BASIC).

Zeichenausgabe mit BSOUT

```
A C000 A9 58          LDA #$58
A C002 20 D2 FF       JSR $FFD2
A C005 00             BRK
```

Beim C16, Plus/4 geben Sie dieses Programm bitte mit der üblichen Startadresse $03F7 ein. Nach dem Start wird an der aktuellen Cursorposition das Zeichen X (ASCII-Code $58 = dezimal 88) geprintet, analog PRINT X. Wenn Sie G C000 schreiben, sieht die Zeile so aus: G C000 X. Dort war der Cursor, wo das X aufgetaucht ist.

Wenn Sie also G C000 schreiben und RETURN drücken lautet die Zeile sofort danach G C000 X. An der Stelle, an der das X aufgetaucht ist, war der CURSOR.

Ebenso wie der PRINT-Befehl kann auch BSOUT Steuerzeichen verwenden. Mit dem ASCII-Code 147 ($93) können Sie den Bildschirm löschen, ohne jede einzelne Speicherzelle des Video-RAM's mit einem Leerzeichen überschreiben zu müssen. Ändern Sie bitte den Befehl LDA #$58 in LDA #$93. Auf die gleiche Weise können Sie beliebige weitere Steuerzeichen verwenden.

*Befehl*: BSOUT ($FFD2)
*Funktion*: Ausgabe des im Akku übergebenen Zeichens oder Steuercodes
*Veränderte Register*: keine

Beispiel (Bildschirm löschen und A ausgeben):

```
A C000 A9 93          LDA #$93
A C002 20 D2 FF       JSR $FFD2
A C005 A9 41          LDA #$41
A C007 20 D2 FF       JSR $FFD2
A C00A 00             BRK
```

## 11.2. Tastatur abfragen mit GETIN

Die Routine GETIN ($FFE4) liest ein Zeichen von der Tastatur ein und übergibt den ASCII-Code (!) des Zeichens im Akkumulator. Wurde keine Taste gedrückt, wird der Code $00 übergeben.

GETIN benötigt keinerlei Parameter, beeinflußt aber alle Register!

Mit einer Kombination aus BSOUT und GETIN können wir eine kleine Textverarbeitung schreiben. Wir benötigen jedoch zwei bisher noch nicht erläuterte Befehle.

*Der BEQ-Befehl*

BEQ ist die Umkehrung von BNE. Auch dieser Befehl prüft, ob die jeweils letzte Operation zum Ergebnis null führte. Im Gegensatz zu BNE wird in diesem Fall jedoch zur angegebenen Adresse verzweigt.

BEQ verzweigt, wenn das Zero-Flag gesetzt ist, also die letzte Operation zum Ergebnis null führte. Ergab sich ein Resultat ungleich null, wird der bedingte Sprung nicht ausgeführt. GETIN und BEQ gestatten somit den Aufbau von Warteschleifen, die erst dann verlassen werden, wenn eine beliebige Taste gedrückt wird.

```
Warteschleife

      A C000 20 E4 FF          JSR $FFE4
      A C003 F0 FB             BEQ $C000
      A C005 00                BRK
```

Der Aufruf von GETIN (JSR $FFE4) liest ein Zeichen von der Tastatur. Wurde keine Taste betätigt, lädt GETIN als letzte Operation den Akkumulator mit dem Wert null. Da die Bedingung "Verzweige, wenn gleich null" (BEQ) erfüllt ist, wird erneut GETIN aufgerufen.

### *Der JMP-Befehl*

Außer den bedingten Sprüngen stellt uns der Rechner einen unbedingten Sprung zur Verfügung, der immer (!) ausgeführt wird, analog dem BASIC-Befehl GOTO. Dem JMP-Befehl folgt die Angabe einer Adresse, zu der verzweigt wird:

```
Endlosschleife

A C000 A9 41            LDA #$41
A C002 20 D2 FF         JSR $FFD2
A C005 4C 00 C0         JMP $C000
```

Im Beispiel wird mit BSOUT ein A ausgegeben, anschließend erfolgt ein unbedingter Sprung zum Programmanfang, der immer ausgeführt wird. Es handelt sich somit um eine Endlosschleife, die niemals beendet wird.

Mit diesen beiden Befehlen können wir unsere Textverarbeitung schreiben:

```
Mini-Textverarbeitung

A C000 20 E4 FF         JSR $FFE4
A C003 F0 FB                BEQ $C000
A C005 20 D2 FF         JSR $FFD2
A C008 4C 00 C0         JMP $C000
```

Die Befehle JSR $FFE4 und BEQ $C000 bilden die erläuterte Eingabeschleife. Wurde eine Taste betätigt, wird das eingegebene Zeichen mit BSOUT auf dem Bildschirm ausgegeben (JSR $FFD2).

Der folgende JMP-Befehl bewirkt die Rückkehr zum Programmanfang. Das Programm läuft daher in einer Endlosschleife, die Zeichen für Zeichen von der Tastatur einliest und ausgibt.

*Routine*: GETIN ($FFE4)
*Funktion*: Liest ein Zeichen von der Tastatur und übergibt den ASCII-Code des Zeichens im Akkumulator ($00 = keine Taste gedrückt)
*Veränderte Register*: Akkumulator, X, Y

```
Beispiel (Auf Taste warten und ausgeben):

A C000 20 E4 FF          JSR $FFE4
A C003 F0 FB                BEQ $C000
A C005 20 D2 FF          JSR $FFD2
```

## 11.3. Cursor setzen mit PLOT

Wie Sie wissen, ist es beim C64 ohne besonderen Aufwand in BASIC-Programmen nicht möglich, den Cursor auf eine bestimmte Bildschirmposition zu setzen, im Gegensatz zu Assembler-Programmen.

Die PLOT-Routine ($FFF0) setzt den Cursor auf eine beliebige Bildschirmposition. Die gewünschte Spalte (0-39) wird im Y-Register und die gewünschte Zeile (0-24) im X-Register übergeben. Die PLOT-Routine beeinflußt ebenso wie GETIN den Inhalt aller Register.

Beachten Sie bitte, daß vor dem Aufruf dieser Routine der uns noch unbekannte Befehl CLC in das Programm einzufügen ist.

*Routine*: PLOT ($FFF0)
*Funktion*: Setzt den Cursor auf die im Y-Register übergebene Spalte und die im X-Register übergebene Zeile. (Voraussetzung: Vor dem Aufruf der Routine muß der Befehl CLC stehen.)
*Veränderte Register*: Akkumulator, X, Y

```
Beispiel (Zeichen A in Spalte 3 von Zeile 5 ausgeben):

A C000 A2 03        LDX #$03
A C002 A0 05        LDY #$05
A C004 18           CLC
A C005 20 F0 FF     JSR $FFF0
A C008 A9 41        LDA #$41
A C00A 20 D2 FF     JSR $FFD2
A C00D 00           BRK
```

## 12. Demoprogramme

Wir stehen vor dem Ende des ersten Hauptteils. Sie kennen nun die wichtigsten Grundlagen der Assembler-Programmierung und verschiedene Betriebssystem-Routinen, die uns die Programmierung sehr erleichtern.

Zur Übung folgen nun einige Demoprogramme, die den praktischen Umgang mit BSOUT, GETIN, PLOT und den bisher erläuterten Assembler-Befehlen aufzeigen.

### 12.1. Spiegelschrift

Das folgende Programm demonstriert den effektiven Umgang mit der indizierten Adressierung. Der Inhalt der obersten Bildschirmzeile wird in der untersten Zeile in Spiegelschrift wiedergegeben.

```
Spiegelschrift: Version 1 (C64)

A C000 A2 27      LDX #$27        ;STARTWERT: DEZIMAL 39
A C002 A0 00      LDY #$00        ;STARTWERT: 0
A C004 BD 00 04   LDA $0400,X     ;$0400 + X LESEN
A C007 99 C0 07   STA $07C0,Y     ;UND NACH $07C0 +Y SCHREIBEN
A C00A C8         INY             ;Y INKREMENTIEREN
A C00B CA         DEX             ;X DEKREMENTIEREN
A C00C D0 F6      BNE $C004       ;X = 0 ? NEIN => VERZWEIGEN
A C00E 00         BRK
```

Geben Sie die im Listing abgebildeten Kommentare bitte nicht mit ein! Der Ablauf: Nachdem das X-Register mit dem Startwert $27 und das Y-Register mit $00 geladen wurden, beginnt die Programmschleife, in der Zeichen für Zeichen gelesen und in die unterste Zeile geschrieben wird.

Im ersten Durchgang wird der Akkumulator mit dem Inhalt der Speicherzelle $0427 ($0400 + X-Inhalt($27)) geladen. Diese Speicherzelle entspricht der letzten Spalte der obersten Bildschirmzeile.

Das Zeichen wird nun nach $07C0 ($07C0 + Y-Inhalt($00)) geschrieben. $07C0 ist die Adresse der ersten Spalte in der untersten Zeile. Das letzte Zeichen der obersten Zeile kommt somit an den Beginn der untersten Zeile.

Im nächsten Durchgang der Schleife soll das vorletzte Zeichen der ersten Zeile zum zweiten Zeichen der untersten Zeile werden. Entsprechend wird das X-Register dekrementiert und das Y-Register inkrementiert (=> 2.Durchgang: Lesen von $0400 + $26); Schreiben nach $07C0 + $01).

Der Ablauf kann folgendermaßen beschrieben werden: Das Programm tastet sich in der obersten Zeile vorwärts und in der untersten rückwärts. Daher werden die Zeichen in - gegenüber dem Original in der ersten Zeile - in umgekehrter Reihenfolge geschrieben, also in Spiegelschrift.

Ein Schönheitsfehler tritt jedoch auf. Die Schleife überträgt nicht 40, sondern nur 39 Zeichen. Das erste Zeichen der obersten Zeile (an Adresse $0400 = $0400 + $00) wird nicht mehr behandelt, da die Schleife verlassen wird, wenn das X-Register den Inhalt $00 besitzt (Verzweigung mit BNE (verzweige, wenn ungleich (!) null)).

Die Korrektur ist jedoch sehr einfach. Wir ändern $0400 in $03FF. Beim letzten Schleifendurchgang besitzt X den Inhalt $01 und es wird, wie gewünscht, auf Adresse $0400 zugegriffen ($0400 = $03FF + $01).

Durch die Korrektur wird jedoch noch eine weitere Korrektur erforderlich. Beim ersten Durchgang wird nicht mehr auf das letzte Zeichen der ersten Zeile zugegriffen (an Adresse $0427), sondern auf das vorletzte, da $03FF + $27 die Adresse $0426 ergibt. Das Problem wird gelöst, indem das X-Register vor Eintritt in die Schleife statt $27 den geänderten Ausgangswert $28 erhält.

```
Spiegelschrift: Version 2 (C64)

A C000 A2 28      LDX #$28        ;STARTWERT: DEZIMAL 40
A C002 A0 00      LDY #$00        ;STARTWERT: 0
A C004 BD FF 03   LDA $03FF,X     ;$03FF + X LESEN
A C007 99 C0 07   STA $07C0,Y     ;UND NACH $07C0 +Y SCHREIBEN
A C00A C8         INY             ;Y INKREMENTIEREN
A C00B CA         DEX             ;X DEKREMENTIEREN
A C00C D0 F6      BNE $C004       ;X = 0 ? NEIN => VERZWEIGEN
A C00E 00         BRK
```

Bei Verwendung dieses Programms auf einem C16, C116 oder Plus/4 müssen Sie außer der geänderten Startadresse $03F7 (und der entsprechend anderen Adresse des Branch-Befehls BNE) auch die Adresse des Bildschirmspeichers berücksichtigen. Da dieser bei $0C00 beginnt, ist der Befehl LDA $03FF,X in LDA $0BFF,X zu ändern.

Die unterste Bildschirmzeile beginnt statt bei $07C0 bei Adresse $0FC0, so daß sich folgendes Programm ergibt:

```
Spiegelschrift (C16, Plus/4)

A 03F7 A2 28      LDX #$28          ;STARTWERT: DEZIMAL 40
A 03F9 A0 00      LDY #$00          ;STARTWERT: 0
A 03FB BD FF 0B   LDA $0BFF,X       ;$0BFF + X LESEN
A 03FE 99 C0 0F   STA $0FC0,Y       ;UND NACH $0FC0 +Y SCHREIBEN
A 0401 C8         INY               ;Y INKREMENTIEREN
A 0402 CA         DEX               ;X DEKREMENTIEREN
A 0403 D0 F6      BNE $03FB         ;X = 0 ? NEIN => VERZWEIGEN
A 0405 00         BRK
```

Gehen Sie bei der Erprobung des Programms so vor:

1. Löschen Sie den Bildschirm und schreiben Sie einen beliebigen Text in die erste Bildschirmzeile (bleiben Sie dabei im Monitor; Sie müssen nicht mit X nach BASIC zurückkehren).

2. Bewegen Sie den Cursor in die zweite oder dritte Zeile und rufen Sie unsere Routine mit G C000 (C64) bzw. G 03F7 (C16, Plus/4) auf.

## 12.2. Rahmen zeichnen

Das folgende Programm demonstriert die Anwendung der PLOT- und der BSOUT-Routine. Um den Bildschirm wird ein Rahmen gelegt, der in der ersten Zeile (Zeile Nr.0) beginnt und in der elften Zeile (Zeile Nr.10) endet. Der Rahmen endet vor der jeweils letzten Spalte einer Zeile. Der Grund: Sie kennen bestimmt das Phänomen, daß Leerzeilen eingefügt werden, wenn man den rechten Bildschirmrand (Spalte 39) beschreibt. Dieses Einfügen wollen wir jedoch vermeiden.

Die Vorgehensweise:

1. In der obersten Bildschirmzeile wird - in einer Schleife - eine Linie gezeichnet, die aus dem Graphikzeichen mit dem ASCII-Code 195 (= $C3) besteht. Vor Beginn der Schleife wird die obere linke Rahmenecke gezeichnet (in Spalte 0), nach der Schleife in Spalte Nummer 38 mit dem entsprechenden Graphikzeichen die obere rechte Ecke gemalt.

2. Die gleiche Linie wird nun in Zeile Nummer 10 gezeichnet, wobei für die untere rechte bzw. linke Ecke entsprechende Graphikzeichen verwendet werden.

3. Ebenfalls in einer Schleife wird die jeweils erste und vorletzte Spalte (vorletzte, um das erwähnte Einfügen von Leerzeilen zu vermeiden) aller zwischen den beiden Linien liegenden Zeilen mit einer senkrechten Linie (ASCII-Code 194 = $C2) beschrieben.

Da dieses Programm doch recht umfangreich ist, wird es abschnittsweise erläutert. Am Ende des Kapitels finden Sie das komplette Programmlisting.

```
Rahmen zeichnen (C64)

Teil 1:              Obere Linie zeichnen
A C000 A0 00         LDY #$00        ;SPALTE 0
A C002 A2 00         LDX #$00        ;ZEILE 0
A C004 18            CLC             ;'PLOT' VORBEREITEN
A C005 20 F0 FF      JSR $FFF0       ;CURSOR MIT PLOT SETZEN
A C008 A9 B0         LDA #$B0        ;$B0 = GRAPHIKZ.OBERE LINKE ECKE
A C00A 20 D2 FF      JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A C00D A2 25         LDX #$25        ;X MIT DEZIMAL 37 LADEN
A C00F A9 C3         LDA #$C3        ;$C3 = GRAPHIKZEICHEN LINIE WAAGR.
A C011 20 D2 FF      JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A C014 CA            DEX             ;SCHLEIFENZÄHLER VERMINDERN
A C015 D0 FA         BNE $C011       ;LINIEN KOMPLETT AUSGEGEBEN?
A C017 A9 AE         LDA #$AE        ;JA, DANN GRAPHIKZ. FÜR OBERE
A C019 20 D2 FF      JSR $FFD2       ;RECHTE ECKE AUSGEBEN
```

Die - nicht einzugebenden (!) - Kommentare zu diesem Teil erklären bereits die wesentlichen Besonderheiten. Zuerst wird der Cursor mit der PLOT-Routine auf die erste Spalte der obersten Zeile gesetzt. Mit BSOUT wird anschließend ein Graphikzeichen ausgegeben, ein Winkel, der die obere linke Ecke des Rahmens darstellen soll.

Nun folgt eine Schleife, in der 37mal das Graphikzeichen waagrechte Linie (ASCII-Code 195 = $C3) ausgegeben wird. Die oberste Zeile ist nun bis auf die vorletzte Spalte komplett behandelt. In dieser wird ebenfalls ein Winkelzeichen ausgegeben, das die obere rechte Ecke des Rahmens darstellt (ASCII-Code 174 = $AE).

```
Teil 2:              Untere Linie zeichnen
A C01C A0 00         LDY #$00        ;SPALTE 0
A C01E A2 0A         LDX #$0A        ;ZEILE 10
A C020 18            CLC             ;'PLOT' VORBEREITEN
A C021 20 F0 FF      JSR $FFF0       ;CURSOR MIT PLOT SETZEN
A C024 A9 AD         LDA #$AD        ;$B0 = GRAPHIKZ.UNTERE LINKE ECKE
A C026 20 D2 FF      JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A C029 A2 25         LDX #$25        ;X MIT DEZIMAL 37 LADEN
A C02B A9 C3         LDA #$C3        ;$C3 = GRAPHIKZEICHEN LINIE WAAGR.
A C02D 20 D2 FF      JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A C030 CA            DEX             ;SCHLEIFENZÄHLER VERMINDERN
A C031 D0 FA         BNE $C02D       ;LINIEN KOMPLETT AUSGEGEBEN?
A C033 A9 BD         LDA #$BD        ;JA, DANN GRAPHIKZ.FÜR UNTERE
A C035 20 D2 FF      JSR $FFD2       ;RECHTE ECKE AUSGEBEN
```

Wie unschwer zu erkennen ist, ist Teil 2 - das Zeichnen der unteren Linie des Rahmens in Zeile 10 - mit Teil 1 nahezu identisch. Der einzige Unterschied besteht in der Verwendung anderer Graphikzeichen für die Eckwinkel.

Davon abgesehen ist der Ablauf unverändert: Der Cursor wird auf Spalte 0 von Zeile 10 positioniert, die linke Ecke gezeichnet, anschließend in einer Schleife 37mal eine waagrechte Linie und zuletzt der rechte Eckwinkel ausgegeben.

```
Teil 3:          Senkrechte Linien zeichnen
A C038 A2 09     LDX #$09        ;SCHLEIFENZÄHLER LADEN
A C03A A0 00     LDY #$00        ;SPALTE = 0
A C03C 18        CLC             ;'PLOT' VORBEREITEN
A C03D 20 F0 FF  JSR $FFF0       ;CURSOR AUF SPALTE 0 VON ZEILE X
A C040 A9 C2     LDA #$C2        ;GRAPHIKZ.SENKRECHTE LINIE
A C042 20 D2 FF  JSR $FFD2       ;AUSGEBEN
A C045 A0 26     LDY #$26        ;SPALTE = $26 = DEZIMAL 38
A C047 18        CLC             ;'PLOT' VORBEREITEN
A C048 20 F0 FF  JSR $FFF0       ;CURSOR AUF SPALTE 38 VON ZEILE X
A C04B A9 C2     LDA #$C2        ;GRAPHIKZ.SENKRECHTE LINIE
A C04D 20 D2 FF  JSR $FFD2       ;AUSGEBEN
A C050 CA        DEX             ;ZEILENZÄHLER VERMINDERN
A C051 D0 E7     BNE $C03A       ;FERTIG? NEIN => VERZWEIGEN
A C053 60        RTS             ;SONST ZURÜCK NACH BASIC
```

Der dritte und letzte Teil unterscheidet sich sehr von den beiden vorangegangen. Innerhalb der Schleife wird der Cursor zweimal gesetzt. Das X-Register dient zugleich als Schleifen- und als Zeilenzähler und erhält zu Beginn den Startwert $09.

Der Cursor wird im ersten Durchgang auf Spalte 0 von Zeile X (also Zeile 9) positioniert und eine senkrechte Linie ausgegeben (ASCII-Code 194 = $C2). Anschließend wird er auf die Spalte $26 (dezimal 38) der gleichen Zeile gesetzt und dort wird ebenfalls eine senkrechte Linie ausgegeben.

Zeile 10 enthält nun die senkrechten Begrenzungen des Rahmens, entsprechende Graphikzeichen in der ersten und der vorletzten Spalte. Nun wird die nächsthöhere Zeile behandelt. Unser Zeilenzähler, das X-Register, wird dekrementiert und ein neuer Schleifendurchgang beginnt, wenn die Bedingung BNE ("verzweige, wenn ungleich null") erfüllt ist, d.h., wenn X nach der Dekrementierung einen beliebigen Wert außer null besitzt.

Auf diese Weise werden alle Zeilen ab Zeile 9 aufwärts behandelt. Wenn der Rahmen komplett ist, wird die Schleife verlassen und ausnahmsweise nicht (!) mit BRK in den Monitor zurückgekehrt.

Diese Routine dürfte in vielen BASIC-Programmen zu gebrauchen sein. Daher endet Sie mit dem Befehl RTS. RTS führt zur Rückkehr in ein BASIC-Programm, wenn dieses die Routine mit SYS 49152 aufrief.

Ein entsprechendes Demoprogramm:

```
100 SYS 49152
110 FOR I=1 TO 1000:NEXT
120 PRINT CHR$(147)
130 SYS 49152
```

Dieses - nicht unbedingt sehr sinnvolle - Demoprogramm ruft die Routine auf, um einen Rahmen zu zeichnen. Nach Ablauf einer Verzögerungsschleife wird der Bildschirm gelöscht und erneut ein Rahmen gezeichnet.

Das komplette Programmlisting (C64):

```
Teil 1:           Obere Linie zeichnen
A C000 A0 00      LDY #$00        ;SPALTE 0
A C002 A2 00      LDX #$00        ;ZEILE 0
A C004 18         CLC             ;'PLOT' VORBEREITEN
A C005 20 F0 FF   JSR $FFF0       ;CURSOR MIT PLOT SETZEN
A C008 A9 B0      LDA #$B0        ;$B0 = GRAPHIKZ.OBERE LINKE ECKE
A C00A 20 D2 FF   JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A C00D A2 25      LDX #$25        ;X MIT DEZIMAL 37 LADEN
A C00F A9 C3      LDA #$C3        ;$C3 = GRAPHIKZEICHEN LINIE WAAGR.
A C011 20 D2 FF   JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A C014 CA         DEX             ;SCHLEIFENZÄHLER VERMINDERN
A C015 D0 FA      BNE $C011       ;LINIEN KOMPLETT AUSGEGEBEN?
A C017 A9 AE      LDA #$AE        ;JA, DANN GRAPHIKZ. FÜR OBERE
A C019 20 D2 FF   JSR $FFD2       ;RECHTE ECKE AUSGEBEN
Teil 2:           Untere Linie zeichnen
A C01C A0 00      LDY #$00        ;SPALTE 0
A C01E A2 0A      LDX #$0A        ;ZEILE 10
A C020 18         CLC             ;'PLOT' VORBEREITEN
A C021 20 F0 FF   JSR $FFF0       ;CURSOR MIT PLOT SETZEN
A C024 A9 AD      LDA #$AD        ;$B0 = GRAPHIKZ.UNTERE LINKE ECKE
A C026 20 D2 FF   JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
```

```
A C029 A2 25      LDX #$25        ;X MIT DEZIMAL 37 LADEN
A C02B A9 C3      LDA #$C3        ;$C3 = GRAPHIKZEICHEN LINIE WAAGR.
A C02D 20 D2 FF   JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A C030 CA         DEX             ;SCHLEIFENZÄHLER VERMINDERN
A C031 D0 FA      BNE $C02D       ;LINIEN KOMPLETT AUSGEGEBEN?
A C033 A9 BD      LDA #$BD        ;JA, DANN GRAPHIKZ.FÜR UNTERE
A C035 20 D2 FF   JSR $FFD2       ;RECHTE ECKE AUSGEBEN
Teil 3 :          Senkrechte Linien zeichnen
A C038 A2 09      LDX #$09        ;SCHLEIFENZÄHLER LADEN
A C03A A0 00      LDY #$00        ;SPALTE = 0
A C03C 18         CLC             ;'PLOT' VORBEREITEN
A C03D 20 F0 FF   JSR $FFF0       ;CURSOR AUF SPALTE 0 VON ZEILE X
A C040 A9 C2      LDA #$C2        ;GRAPHIKZ.SENKRECHTE LINIE
A C042 20 D2 FF   JSR $FFD2       ;AUSGEBEN
A C045 A0 26      LDY #$26        ;SPALTE = $26 = DEZIMAL 38
A C047 18         CLC             ;'PLOT' VORBEREITEN
A C048 20 F0 FF   JSR $FFF0       ;CURSOR AUF SPALTE 38 VON ZEILE X
A C04B A9 C2      LDA #$C2        ;GRAPHIKZ.SENKRECHTE LINIE
A C04D 20 D2 FF   JSR $FFD2       ;AUSGEBEN
A C050 CA         DEX             ;ZEILENZÄHLER VERMINDERN
A C051 D0 E7      BNE $C03A       ;FERTIG? NEIN => VERZWEIGEN
A C053 60         RTS             ;SONST ZURÜCK NACH BASIC
```

Geben Sie die Kommentare bitte nicht ein, sondern wie üblich nur die Assembler-Befehle selbst.

Nachstehend ist das komplette Listing für den C16, Plus/4 abgebildet. Geben Sie das Programm bitte nicht wie sonst ab $03F7 ein! Der bisher zur Programmablage verwendete Bereich $03F7-$0436 ist für dieses umfangreiche Programm zu klein.

Wir verwenden den Bereich $065E-$06EB, der - im Normalfall unbenutzten - RAM-Bereich für einen Sprachsynthesizer. Rufen Sie das Programm entsprechend von BASIC aus mit SYS 1630 auf (1630 = $065E).

Das komplette Programmlisting (C16, Plus/4):

```
Teil 1:          Obere Linie zeichnen
A 065E A0 00     LDY #$00        ;SPALTE 0
A 0660 A2 00     LDX #$00        ;ZEILE 0
A 0662 18        CLC             ;'PLOT' VORBEREITEN
A 0663 20 F0 FF  JSR $FFF0       ;CURSOR MIT PLOT SETZEN
A 0666 A9 B0     LDA #$B0        ;$B0 = GRAPHIKZ.OBERE LINKE ECKE
A 0668 20 D2 FF  JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A 066B A2 25     LDX #$25        ;X MIT DEZIMAL 37 LADEN
A 066D A9 C3     LDA #$C3        ;$C3 = GRAPHIKZEICHEN LINIE WAAGR.
A 066F 20 D2 FF  JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A 0672 CA        DEX             ;SCHLEIFENZÄHLER VERMINDERN
A 0673 D0 FA     BNE $066F       ;LINIEN KOMPLETT AUSGEGEBEN?
A 0675 A9 AE     LDA #$AE        ;JA, DANN GRAPHIKZ. FÜR OBERE
A 0677 20 D2 FF  JSR $FFD2       ;RECHTE ECKE AUSGEBEN
Teil 2:          Untere Linie zeichnen
A 067A A0 00     LDY #$00        ;SPALTE 0
A 067C A2 0A     LDX #$0A        ;ZEILE 10
A 067E 18        CLC             ;'PLOT' VORBEREITEN
A 067F 20 F0 FF  JSR $FFF0       ;CURSOR MIT PLOT SETZEN
A 0682 A9 AD     LDA #$AD        ;$B0 = GRAPHIKZ.UNTERE LINKE ECKE
A 0684 20 D2 FF  JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A 0687 A2 25     LDX #$25        ;X MIT DEZIMAL 37 LADEN
A 0689 A9 C3     LDA #$C3        ;$C3 = GRAPHIKZEICHEN LINIE WAAGR.
A 068B 20 D2 FF  JSR $FFD2       ;MIT 'BSOUT' AUSGEBEN
A 068E CA        DEX             ;SCHLEIFENZÄHLER VERMINDERN
A 068F D0 FA     BNE $068B       ;LINIEN KOMPLETT AUSGEGEBEN?
A 0691 A9 BD     LDA #$BD        ;JA, DANN GRAPHIKZ.FÜR UNTERE
A 0693 20 D2 FF  JSR $FFD2       ;RECHTE ECKE AUSGEBEN
Teil 3:          Senkrechte Linien zeichnen
A 0696 A2 09     LDX #$09        ;SCHLEIFENZÄHLER LADEN
A 0698 A0 00     LDY #$00        ;SPALTE = 0
A 069A 18        CLC             ;'PLOT' VORBEREITEN
A 069B 20 F0 FF  JSR $FFF0       ;CURSOR AUF SPALTE 0 VON ZEILE X
A 069E A9 C2     LDA #$C2        ;GRAPHIKZ.SENKRECHTE LINIE
A 06A0 20 D2 FF  JSR $FFD2       ;AUSGEBEN
A 06A3 A0 26     LDY #$26        ;SPALTE = $26 = DEZIMAL 38
A 06A5 18        CLC             ;'PLOT' VORBEREITEN
A 06A6 20 F0 FF  JSR $FFF0       ;CURSOR AUF SPALTE 38 VON ZEILE X
```

```
A 06A9 A9 C2      LDA #$C2        ;GRAPHIKZ.SENKRECHTE LINIE
A 06AB 20 D2 FF   JSR $FFD2       ;AUSGEBEN
A 06AE CA         DEX             ;ZEILENZÄHLER VERMINDERN
A 06AF D0 E7      BNE $0698       ;FERTIG? NEIN => VERZWEIGEN
A 06B1 60         RTS             ;SONST ZURÜCK NACH BASIC
```

# Teil 2: Fortgeschrittene Assembler-Programmierung

Sie kennen nun alle Grundlagen der Assembler-Programmierung und sollten sich erst einmal mit der selbständigen Erstellung weiterer Übungsprogramme beschäftigen.

Gerade für die relativ komplexen Themen Schleifenbildung und indirekt-indizierte Adressierung gilt, daß diese nur durch Übung wirklich zu verstehen sind. Experimentieren Sie also und lassen Sie sich durch gelegentliche Abstürze nicht irritieren.

Zum Thema Abstürze: bei der Assembler-Programmierung ist ein sogenannter RESET-Schalter eine absolute Notwendigkeit. Ein solcher Schalter ist beim C16, C116, Plus/4 und C128 bereits eingebaut.

C64-Besitzer können jederzeit einen RESET-Schalter erwerben, der auf den USER-Port oder einen Floppy-Anschluß aufgesteckt wird.

Wie Sie sicher feststellten, ist es bei der Assembler-Programmierung öfters der Fall, daß sich der Rechner aufhängt und nicht zur Weiterarbeit zu bewegen ist. Vor allem bei der in diesem Aufbaukurs besprochenen Interrupt-Programmierung wird dies öfters geschehen.

Mit einem RESET-Schalter ist der Rechner wieder zum Leben zu erwecken, wobei, im Gegensatz zum Ausschalten, Assembler-Programme meist unverändert erhalten bleiben.

Da im folgenden Teil immer weniger Programme in einer eigenen C16, Plus/4-Version vorgestellt werden, will ich die Besitzer eines dieser Rechner kurz an die nötigen Änderungen erinnern:

- Verwenden Sie statt $C000 die Startadresse $03F7.

- Ändern Sie entsprechend die Sprungadressen von JMP- und Branch-Befehlen wie BNE, so daß der Sprung zum entsprechenden Befehl wie in der C64-Version führt.

## 13. Schleifen unter die Lupe genommen

Die vorangegangenen Kapitel setzten Sie bereits in die Lage, kleine Assembler-Programme zu schreiben. Sie kennen die Transfer-Befehle, die In-/Dekrementierbefehle, die Branch-Befehle BEQ und BNE und, die Befehle JSR und JMP, und die wichtigsten Adressierungsarten.

Da Sie bestimmt auf den Geschmack gekommen sind und eigene Übungsprogramme geschrieben haben, besitzen Sie inzwischen handfeste Assembler-Grundkenntnisse. Auf den folgenden Seiten werden wir daher mit der Verwirklichung anspruchsvollerer Programmprojekte beginnen, für die wir jedoch weitere Befehle benötigen.

Da diese Projekte höhere Anforderungen an Sie stellen werden, hole ich zuerst einige Versäumnisse nach und nehme Befehle, die bisher nur oberflächlich besprochen wurden, genauer unter die Lupe. Auf den folgenden Seiten wird vorwiegend die Schleifenbildung behandelt, wobei außer den bereits bekannten Befehlen BEQ und BNE weitere Befehle zur Schleifenbildung eingeführt werden.

### 13.1. Näheres zu den In- und Dekrementierbefehlen

Angenommen, eine Speicherzelle oder ein Register besitzt bereits dem Maximalwert $FF und wird nun inkrementiert. Oder aber, umgekehrt, ein Register hat den Inhalt $00 und wird nun dekrementiert.

Wissen Sie, was in einem solchen Fall passiert? Wenn nicht, führen Sie mit mir einige kleine Experimente durch (C16- und Plus-Besitzer geben die folgenden Programme bitte jeweils mit der Startadresse $03F7 ein und rufen sie entsprechend mit G 03F7 auf):

```
A C000 A2 FF          LDX #$FF
A C002 E8             INX
A C003 00             BRK
```

Wir laden das X-Register mit $FF und inkrementieren es anschließend. Die Registeranzeige des Monitors sollte anschließend etwa so aussehen:

```
.....AC XR YR .....
.....00.....
```

Das X-Register enthält den Wert $00. Eines der angesprochenen Geheimnisse wäre damit gelöst. Wenn eine Speicherzelle inkrementiert wird, die bereits den höchsten Wert $FF enthält, erhält Sie dadurch den niedrigsten Wert $00.

```
A C000 A2 00          LDX #$00
A C002 CA             DEX
A C003 00             BRK
```

Das zweite Geheimnis deckt dieses Programm auf. Die Registeranzeige:

```
.....AC XR YR .....
.....FF.....
```

Somit steht ebenfalls fest, daß eine Speicherzelle, die den Inhalt $00 besitzt und nun dekrementiert wird, den Wert $FF erhält. Immer dann, wenn ein Ende der Werteskala erreicht ist, der größte oder aber der kleinste mögliche Wert, beginnt die Zählung wieder beim entgegengesetzten Ende.

Diese Feststellung bezieht sich keineswegs ausschließlich auf das X-Register, sondern auf alle Speicherzellen und Register!

## 13.2. Ein weiteres Mal: BNE und BEQ

Wie die vorhergehenden Kapitel zeigten, setzen uns erst Schleifen in die Lage, sinnvolle Assembler-Programme zu schreiben. Sie kennen nun bereits zwei Schleifenbefehle, die bedingten Sprungbefehle BNE (branch if not equal zero, verzweige, wenn ungleich null) und BEQ (branch if equal zero, verzweige, wenn gleich null), wobei sich der Ausdruck gleich null bzw. ungleich null immer auf das Ergebnis der letzten Operation bezieht, die das Zero-Flag beeinflußte (z.B. Transferbefehle oder In-/Dekrementierbefehle).

Um die Schleifenprogrammierung auf die vielfältigsten Probleme anwenden zu können, benötigen wir weitere Befehle. Der Standardbefehl BNE besitzt einen gravierenden Nachteil. Wenn der Schleifenzähler bis zum Wert null dekrementiert ist, wird die Schleife verlassen. Daher war es uns zuvor nicht möglich, bei der Ausgabe des Zeichensatzes auch das Zeichen mit dem Code $00 - den Klammeraffen - auszugeben.

Rufen wir uns kurz in Erinnerung, wie die Befehle BEQ und BNE ablaufen. Die Transfer-Befehle (STA, STX, TAX, TYA...) und die In-/Dekrementierbefehle (INC, DEX...) beeinflussen ein Bit des Status-Registers, das sogenannte Zero-Flag. Führte einer der genannten Befehle zum Ergebnis null (wurde z.B. der Akku mit dem Wert $00 geladen oder eine Speicherzelle dekrementiert, die zuvor den Inhalt eins besaß), wird automatisch das Zero-Flag gesetzt, ansonsten gelöscht. Die folgende Skizze veranschaulicht den Aufbau des Status-Registers, wobei uns von den verschiedenen Flags jedoch vorläufig nur das Zero-Flag interessieren soll.

```
Die Bits (=Flags) des Status-Registers

  7      6          5        4      3        2         1     0
Negativ Überlauf (unbenutzt) Break Dezimal Interrupt Zero Carry
```

BEQ und BNE prüfen den Zustand dieses Flags (gesetzt oder gelöscht), um festzustellen, ob die letzte Operation, die das Flag beeinflußte, zum Ergebnis null führte. Wenn ja, verzweigt BEQ zur angegebenen Adresse, ansonsten wird der folgende Programmbefehl bearbeitet.

Wie wir wissen, arbeitet BNE genau umgekehrt. Solange die letzte Operation nicht den Wert null ergab, wird zur angegebenen Adresse verzweigt, ansonsten wird das Programm mit dem nächsten Befehl fortgesetzt.

Diese Erkenntnisse sollen anhand mehrerer Demoprogramme vertieft werden (C16, Plus-Besitzer geben die Programme wie üblich mit der Startadresse $03F7 ein):

```
A C000 A9 00          LDA #$00
A C002 00             BRK
```

Nach dem Aufruf mit G C000 betrachten Sie bitte die Registeranzeige des Monitors, genauer: den Inhalt des Status-Registers (SR). Wie üblich wird eine zweistellige Hexadezimalzahl ausgegeben.

Wenn Sie sich die Mühe machen und die ausgegebene Hexadezimalzahl in eine Dualzahl umzuwandeln, werden Sie feststellen, daß das Bit 1 (Stellenwert von Bit 1: $02) dieser Zahl gesetzt ist, das Zero-Flag.

Die einfachste Methode zur Umwandlung einer Hexadezimalzahl in eine Dualzahl verläuft nach folgendem Schema:

1. Gegeben sei eine zweistellige Hexadezimalzahl, z.B. $7F.
2. Wandeln Sie die rechte Ziffer in eine vierstellige Dualzahl um. $F wird zu %1111.
3. Wandeln Sie die linke Stelle entsprechend um. $7 wird zu %0111. Schreiben Sie diese Dualzahl links neben die zuerst ermittelte Dualzahl.
4. Als Ergebnis erhalten Sie im Beispiel $7F = %01111111.

Ein weiteres Beispiel: $3A = $3 und $A = %0011 und %1010 = %00111010. Um zu erkennen, ob das Zero-Flag des Status-Registers - also Bit 1 - gesetzt oder gelöscht ist, reicht es sogar aus, die rechte Ziffer der Hexadezimalzahl - die den Bits 0-3 entspricht - in eine vierstellige Dualzahl umzuwandeln.

In dieser rechten Ziffer ist nach Ablauf des Programms das Bit 1 - das Zero-Flag - tatsächlich gesetzt, da die Operation LDA #$00 eine null als Ergebnis hat. Die Gegenprobe:

```
A C000 A9 01          LDA #$01
A C000 00             BRK
```

Die dem BRK-Befehl folgende Registeranzeige gibt das Status-Register wiederum als zweistellige Hex-Zahl aus. Das Bit 1 dieser Zahl ist diesmal jedoch gelöscht, wie erwartet, da der Akkumulator mit einem Wert ungleich null geladen wurde.

## 13.3. Schleifen mit BPL und BMI

Bevor ich weitere Schleifenbefehle einführe, zeige ich Ihnen ein kleines Phänomen. Geben Sie bitte ein:

```
A C000 A2 7A          LDX #$7A
A C002 E8             INX
A C003 00             BRK
```

Rufen Sie das Programm wieder mit G C000 auf (C16, Plus/4-Besitzer geben dieses Programm bitte wie üblich ab $03F7 ein und rufen es auch mit G 03F7 auf). Schauen Sie sich nun die Registeranzeige des Monitors an:

```
.....SR AC XR.....
.....3?....7B.....
```

Uns interessiert im Moment nur die Anzeige des X- und des Status-Registers. Das X-Register enthält natürlich den Wert $7B, da es mit dem Wert $7A geladen und anschließend durch INX um eins erhöht wurde.

Vom Status-Register interessiert uns nur Bit Nummer 7. Bit 7 ist das Bit ganz links in einer achtstelligen Dualzahl und somit in der linken Ziffer der Hex-Zahl enthalten. Für diese Ziffer ergibt sich nach dem gezeigten Schema %0011 (wenn $30 ausgegeben wurde).

Das Bit ganz links (Bit 7) ist somit gelöscht. Dieses Bit nennt man Negativ-Flag oder N-Flag. Das N-Flag besitzt ebenso wie das Zero-Flag eine spezielle Bedeutung, die wir nun erforschen wollen.

Versuchen Sie nun folgendes: starten Sie das Programm viermal nacheinander mit G C002 (G 03F9), daß heißt ab dem Befehl INX, und betrachten Sie sich jeweils die Registeranzeige (X-Register und Statusregister).

Der Inhalt des X-Registers wird sich nach jedem Aufruf um eins erhöhen ($7C, $7D, $7E, $7F), während sich das Status-Register nicht verändert.

Nun passen Sie gut auf. Rufen Sie das Programm ein weiteres Mal mit G C002 auf. Das X-Register enthält nun den Wert $80. Soweit ist alles in Ordnung. Unerklärlich ist jedoch, warum sich auf einmal das Status-Register verändert, um genau zu sein, warum sich nur die linke Ziffer ändert.

Anstatt $30 wird auf einmal $B0 ausgegeben. Wenn wir diese Zahl bitweise untersuchen, stellen wir fest, daß Bit 7, das N-Flag, auf einmal gesetzt ist, während es zuvor die ganze Zeit gelöscht war.

Je nach verwendetem Monitor kann die Anzeige des Status-Registers variieren. Das Negativ-Flag (Bit 7) befindet sich jedoch auf alle Fälle im beschriebenen Zustand.

Wenn Sie wollen, können Sie das Programm solange aufrufen, bis das X-Register den Wert $FF enthält und beim nächsten Aufruf der Überschlag nach $00 erfolgt. Sie werden feststellen, daß in diesem Moment das N-Flag wieder gelöscht wird.

Sie können natürlich auch LDX #$FF eingeben und das Programm dann nur einmal starten.

Fassen wir unsere Untersuchungsergebnisse zusammen: immer dann, wenn der Befehl INX zu einem Resultat führt, das kleiner ist als $80 (=dezimal 128), wird das N-Flag gelöscht.

Führt INX zu einem Ergebnis von 128 oder größer, wird das Negativ-Flag gesetzt.

Ohne weitere Untersuchung will ich Ihnen verraten, daß nicht nur der INX-Befehl, sondern auch DEX, INY, DEY, INC, DEC das N-Flag auf die gleiche Weise beeinflussen. Außer diesen In-/Dekrementier-Befehlen wird das N-Flag unter anderem auch von allen besprochenen Ladebefehlen beeinflußt, also von LDA, LDX und LDY.

Diesen Effekt können wir für den Schleifenbau ausnutzen. Es existieren zwei relative Sprungbefehle, die den Zustand des N-Flags testen und je nach Ergebnis zur angegebenen Adresse verzweigen oder nicht verzweigen.

BPL (branch if plus, verzweige, wenn positiv) verzweigt zur angegebenen Adresse, wenn das Negativ-Flag nicht gesetzt ist. BMI (branch if minus, verzweige, wenn negativ) verzweigt im entgegengesetzten Fall, wenn das N-Flag gesetzt ist.

```
A C000 A2 10        LDX #$10
A C002 CA           DEX
A C003 10 FD        BPL $C002  (C16, Plus/4: BPL $03F9)
A C005 00           BRK
```

Wenn Sie dieses Programm aufrufen, werden Sie anschließend feststellen, daß das X-Register den Wert $FF enthält. Diese Schleife wurde somit auch dann noch einmal durchlaufen, als das X-Register bereits den Wert $00 enthielt. Erst bei der folgenden Dekrementierung, die den Wert $FF ergab, wurde die Schleife verlassen.

Warum? Nun, solange X von $10 bis $00 heruntergezählt wurde, war das N-Flag gelöscht und die Bedingung "verzweige, wenn positiv" erfüllt, der bedingte Sprung wurde ausgeführt.

In dem Moment, in dem das X-Register von $00 auf $FF dekrementiert wurde, der DEX-Befehl somit zu einem Ergebnis größer als $79 führte, wurde das Negativ-Flag gesetzt und die Bedingung "verzweige, wenn positiv" war nicht länger erfüllt. Die folgende Tabelle gibt diese Zusammenhänge wieder:

| Verzweigung | letzte Operation | Zero-Flag | Negativ-Flag |
|---|---|---|---|
| BEQ | gleich null | 1 | |
| BNE | ungleich null | 0 | |
| BMI | größer $79 | | 1 |
| BPL | kleiner $80 | | 0 |

Ein Beispiel zum Lesen der Tabelle: BEQ verzweigt, wenn das Ergebnis der letzten Operation den Wert null ergab (Zero-Flag gesetzt).

Die Unterschiede zwischen BNE und BPL bei der Schleifenbildung demonstriert folgendes Programm:

```
Vergleich: BNE und BPL (C64)

A C000 A2 80            LDX #$20
A C002 8A               TXA
A C003 9D 00 04         STA $0400,X
A C006 CA               DEX
A C007 10 F9            BPL $C002
A C009 00               BRK
```

```
Vergleich: BNE und BPL (C16, Plus/4)

A 03F7 A2 80          LDX #$20
A 03F9 8A             TXA
A 03FA 9D 00 0C       STA $0C00,X
A 03FD CA             DEX
A 03FE 10 F9          BPL $03F9
A 0400 00             BRK
```

Das Programm ist altbekannt. Es gibt den Zeichensatz Ihres Rechners auf dem Bildschirm aus. Diesmal wird jedoch der Branch-Befehl BNE durch BPL ersetzt. Den Unterschied sehen Sie nach dem Aufruf mit G C000 (G 03F7). Im Gegensatz zur alten Version wird auch das Zeichen mit dem Code $00 ausgegeben, der Klammeraffe.

Mit BPL gebildete Schleifen werden wie erläutert auch dann noch einmal durchlaufen, wenn das Zählregister den Wert $00 besitzt. Der Nachteil des Programms: da das X-Register nicht mit einem Wert geladen werden darf, der größer als $80 ist, wird nur die erste Hälfte des Zeichensatzes ausgegeben, die Zeichen mit den Codes $81-$FF fehlen.

Zusammenfassung:

1. Zur Schleifenbildung werden vorwiegend die Branch-Befehle BEQ ("verzweige, wenn gleich null") und BNE ("verzweige, wenn ungleich null") bzw. BPL ("verzweige, wenn positiv") und BMI ("verzweige, wenn negativ") eingesetzt. Ist die betreffende Bedingung erfüllt, wird zur angegebenen Adresse verzweigt, ansonsten das Programm mit dem folgenden Befehl fortgesetzt (Vergleiche: IF ... THEN GOTO ...).

2. BEQ und BNE testen das Zero-Flag und stellen am Zustand dieses Flags (gesetzt/gelöscht) fest, ob die letzte Operation zum Ergebnis null führte oder nicht (Ergebnis null: Zero-Flag gesetzt; Ergebnis ungleich null: Zero-Flag gelöscht).

3. BPL und BMI testen das Negativ-Flag und stellen am Flag-Zustand fest, ob die letzte Operation zu einem positiven oder aber zu einem negativen Ergebnis führte. Positiv und Negativ kennzeichnen hierbei die Größe des Ergebnisses. Bleibt das Ergebnis unter $80, wird es als positiv (Negativ-Flag gelöscht), darüber als negativ (Negativ-Flag gesetzt) bezeichnet. Ein Ergebnis wird daher als negativ bezeichnet, wenn Bit 7 (Wert $80) gesetzt ist.

4. Zum Einsatz von BNE und BPL in Schleifen: mit BNE wird eine Schleife verlassen, wenn der Schleifenzähler dekrementiert wird und anschließend den Wert $00 enthält (das Zero-Flag gesetzt wird). Mit BPL wird eine Schleife erst dann verlassen, wenn die Dekrementierung zum Überschlag von $00 zu $FF führt, das Ergebnis also negativ wird. Die BPL-Schleife besitzt daher genau einen Durchgang mehr als die BNE-Schleife.

5. Aus 4. geht hervor, daß das Zählregister bei der BNE-Schleife im letzten Durchgang den Wert $01 besitzt, bei der BPL-Schleife dagegen den Wert $00.

6. Mit BPL gebildete Schleifen können maximal 129mal durchlaufen werden, nämlich dann, wenn das Zählregister mit $80 initialisiert wird. Wird es mit einem größeren Wert geladen, passiert folgendes: das X-Register enthält z.B. den Wert $90. Der erste DEX-Befehl führt nun zum Ergebnis $8F. Da dieses Ergebnis größer ist als 127, wird das Negativ-Flag gesetzt, um ein negatives Ergebnis anzuzeigen, und die Bedingung BPL ("verzweige, wenn positiv") ist nicht erfüllt!

7. BNE, BEQ, BPL und BMI können natürlich nur dann zum Testen und eventuellen Verzweigen eingesetzt werden, wenn die interessierende Operation (in Schleifen meist DEX, DEY, INX und INY) auch tatsächlich den Flag-Zustand je nach Ergebnis verändert. Folgende Befehlsklassen, die wir kennen, beeinflussen Zero- und Negativ-Flag: alle In-/Dekrementier-Befehle (DEC, INX...), alle Lade-Befehle (LDA, LDY...), alle Register-Transfer-Befehle

(TAX, TYA...). Nach jedem dieser Befehle können die erläuterten Branch-Befehle für bedingte Sprünge eingesetzt werden.

Welche Befehle welche Flags beeinflussen, beschreibt die Befehlsübersicht im Anhang.

## 13.4. Sprungweite der Branch-Befehle

Vielleicht ist Ihnen bereits aufgefallen, daß die in den Branch-Befehlen eingegeben Adressen in sehr merkwürdiger Art und Weise codiert werden. Der im letzten Beispiel verwendete Branch-Befehl wurde in folgender Form umgesetzt:

```
A 03FE 10 F9            BPL $C002
```

Die Zahlen 10 und F9 werden beim hinten abgedruckten Monitor nicht angezeigt. Benutzen Sie gegebenenfalls den Befehl

```
M 03FE
```

Anstelle der angegebenen Zwei-Byte-Adresse folgt dem Befehlscode für BPL ($10) das Byte $F9. Branch-Befehle sind sogenannte relative Sprungbefehle. Anstelle der angegebenen Adresse legt der Monitor die Entfernung (!) zu dieser Adresse im Speicher ab.

Nun beträgt die Entfernung im obigen Beispiel sicher weniger als $F9 (=dezimal 249) Byte. Geben Sie bitte zum Vergleich das folgende Programm ein.

```
A C000 A2 FF            LDX #$FF
A C002 D0 03            BNE $C007
A C004 8E 00 04         STX $0400
A C007 00               BRK
```

In diesem (völlig sinnlosen) Programm erfolgt ein Vorwärtssprung zum Befehl BRK an Adresse $C007. Die Entfernung zu diesem Befehl wird mit dem Byte $03 angegeben.

Um diese Angabe zu verstehen, müssen Sie sich die Befehlsausführung in Erinnerung rufen. Der Befehlscode und die Adresse werden gelesen, wobei der Programmzähler jeweils inkrementiert wird.

Nach dem Einlesen des Befehls BNE $C007 weist der Programmzähler somit auf den Befehlscode des folgenden Befehls (STX $C000) und enthält daher die Adresse $C004. Die Angabe der Sprungweite bezieht sich auf diese Adresse, von der aus die Entfernung bis zur gewünschten Adresse $C007 genau drei Byte beträgt. Merken Sie sich daher bitte, daß die Angabe der Sprungweite immer von der Adresse des dem Branch-Befehl folgenden (!) Befehls ausgeht.

Nun zu der seltsam großen Sprungweite $F9 aus dem vorhergehenden Beispiel. In der Angabe der Sprungweite muß eine Information darüber enthalten sein, ob es sich um einen Vorwärts- oder um einen Rückwärtssprung handelt. Diese Information befindet sich im siebten Bit, das bekanntlich den Wert $80 (=dezimal 128) besitzt. Sprungweitenbytes, die größer sind als $80, kennzeichnen somit einen Rückwärtssprung.

Diese relative Adressierung ist zwar platzsparend, da zur Angabe der Zieladresse ein Byte genügt, besitzt jedoch einen Nachteil. Zur Kennzeichnung eines relativen Sprungs stehen pro Richtung nur 128 verschiedene Zahlen zur Verfügung ($00-$7F für Vorwärtssprünge; $80-$FF für Rückwärtssprünge).

Relative Sprünge sind daher auf einen Umkreis von 128 Byte beschränkt. In einem großen Programm ist es leider nicht möglich, mit einem relativen Sprung von einem Ende des Programms zum anderen zu gelangen, im Gegensatz zum JMP-Befehl, mit dem zu beliebigen (!) Adressen verzweigt werden kann, unabhängig von deren Entfernung.

Da Assembler-Programme jedoch sehr kompakt und kurz sind, reicht diese maximale Sprungweite in der Praxis für alle anfallenden Probleme völlig aus.

Zusammenfassung:

1. Alle Branch-Befehle arbeiten mit der relativen Adressierung, bei der anstelle der absoluten Adresse des Sprungziels die Entfernung zu der betreffenden Adresse im Speicher abgelegt wird.

2. Relative Sprünge sind nach vorn (in Richtung höherer Adressen) und nach hinten (in Richtung niedrigerer Adressen) möglich. Rückwärtssprünge werden durch ein gesetztes siebtes Bit gekennzeichnet.

3. Ein Vorteil der relativen Adressierung besteht in der Kürze der Adreßangabe. Die Entfernung wird im Gegensatz zur absoluten Adressierung (zwei Byte) mit nur einem Byte angegeben.

4. Der Nachteil der relativen Adressierung liegt in der beschränkten Sprungweite. Mit einem Branch-Befehl kann nur zu einer maximal 128 Byte entfernten Adresse verzweigt werden.

## 13.5. Übungsprogramme zur Schleifenbildung

Die letzten Kapitel boten einiges über Branch-Befehle und Schleifenbildung. Um es Ihnen zu ermöglichen, das angesammelte Wissen auch tatsächlich zu verdauen", werden wir in diesem Abschnitt einige Programme verwirklichen, die alle mit Schleifen arbeiten.

### 13.5.1. Ball bewegen

Dieses erste Programmprojekt ähnelt einem früher vorgestellten Programm, das einen Ball über eine Bildschirmzeile bewegte.

Diesmal soll der Ball jedoch quer von rechts unten nach links oben (Spalte 23, Zeile 23, dann Spalte 22 und Zeile 22, zum Schluß Spalte 1, Zeile 1) über den gesamten Bildschirm bewegt

werden. Außerdem soll die Bewegung erst fortgesetzt werden, wenn Sie eine beliebige Taste betätigen.

Um den Ball an der jeweils gewünschten Position auszugeben, setzen wir zuerst den Cursor mit der PLOT-Routine des Betriebssystems. Die Ausgabe des ASCII-Codes 113 ($71) malt das für den Ball verwendete Zeichen an der aktuellen Cursorposition.

Bevor der Ball an dieser Position durch Ausgabe eines Leerzeichens (ASCII-Code 32 = $20) wieder gelöscht wird, warten wir mit der GETIN-Routine auf eine Tastenbetätigung.

Lassen Sie sich von der Länge des folgenden Programm-Listings nicht abschrecken. Das Programm ist zwar recht lang, aber dennoch ohne größere Probleme zu verstehen. Bei genauem Studium des Listings lernen Sie einiges über den effektiven Umgang mit den Registern.

```
Ball bewegen (C64)

A C000 A9 93      LDA #$93      ;ASCII-CODE 147 AUSGEBEN
A C002 20 D2 FF   JSR $FFD2     ;=> BILDSCHIRM LOESCHEN
A C005 A2 17      LDX #$17      ;ZEILENZAEHLER INITIAL.

A C007 8A         TXA           ;ZEILE NACH AKKU
A C008 A8         TAY           ;AKKU NACH Y-REGISTER
A C009 18         CLC
A C00A 20 F0 FF   JSR $FFF0     ;CURSOR SETZEN

A C00D A9 71      LDA #$71      ;ASCII-CODE FUER BALL
A C00F 20 D2 FF   JSR $FFD2     ;AUSGEBEN
A C012 86 FB      STX $FB       ;ZEILE (X) 'RETTEN'

A C014 20 E4 FF   JSR $FFE4     ;TASTATUR ABFRAGEN
A C017 F0 FB      BEQ $C014     ;TASTE GEDRUECKT?

A C019 A6 FB      LDX $FB       ;ZEILE WIEDERHOLEN NACH X
A C01B 8A         TXA           ;ZEILE NACH AKKU
A C01C A8         TAY           ;AKKU NACH SPALTE
```

```
A C01D 18          CLC
A C01E 20 F0 FF    JSR $FFF0      ;CURSOR SETZEN

A C021 A9 20       LDA #$20       ;ASCII-CODE F.LEERZEICHEN
A C023 20 D2 FF    JSR $FFD2      ;AUSGEBEN

A C026 CA          DEX            ;ZEILENZAEHLER DEKREMENT.
A C027 D0 DE       BNE $C007      ;FERTIG?
A C029 00          BRK            ;PROGRAMMENDE
```

Der Übersicht wegen sind im Listing Leerzeilen zwischen den verschiedenen Programmteilen eingefügt. Geben Sie diese Leerzeilen bitte nicht mit ein!

Zum Programmablauf: der erste - vorbereitende - Programmteil löscht den Bildschirm, indem mit BSOUT der ASCII-Code 147 ($93) ausgegeben wird. Dieser Code entspricht laut ASCII-Tabelle der Taste CLR, die bekanntlich ebenfalls den Bildschirm löscht. Der erste Programmteil lädt anschließend das X-Register mit dem Wert $17 (=dezimal 23). Das X-Register gibt im weiteren Programmverlauf die Zeile an, in der der Ball auszugeben ist.

Wie erläutert, soll die Ballbewegung ab Position 23/23 (Spalte/Zeile) beginnen. Spalte und Zeile sind somit identisch. Daher wird der Inhalt des Zeilenzählers X in das Y-Register übertragen, bevor mit der PLOT-Routine der Cursor auf die angegebene Position gesetzt wird (X-Register=Zeile; Y-Register=Spalte).

Leider gibt es keinen Befehl, mit dem der Inhalt des X-Registers direkt in das Y-Registers kopiert werden kann. Daher nehmen wir einen Umweg über den Akkumulator. Mit TXA wird der Inhalt in den Akkumulator kopiert, und anschließend mit TAY von dort aus ins Y-Register weiterbefördert.

Sowohl X- als auch Y-Register enthalten nun den Wert $17 und der Cursor kann auf diese Position gesetzt werden (CLC und JSR $FFF0).

Nun wird der ASCII-Code des Balls ausgegeben ($71). Der Ball erscheint in Spalte 23 von Zeile 23. Bevor mit der GETIN-Routine auf eine Taste gewartet wird, kopiert der Befehl STX $FB den Inhalt des X-Registers in die Zeropage-Speicherzelle $FB.

Der Grund: wie erläutert (siehe Betriebssystem-Routinen), verändert die GETIN-Routine die Inhalte der Register. Der aktuelle Wert unseres Zeilenzählers muß jedoch unbedingt erhalten bleiben! Dieser Wert wird daher in die Speicherzelle $FB gerettet.

Die bereits bekannte Warteschleife ruft immer wieder GETIN auf, solange keine Taste betätigt wurde. Wie erläutert, ist die Bedingung "verzweige, wenn gleich null" solange erfüllt, bis eine Taste betätigt wird (siehe GETIN-Routine).

Erfolgte eine beliebige Tastenbetätigung, wird der alte Inhalt des X-Registers zurückgeholt. Das X-Register wird mit dem zuvor nach $FB kopierten Wert geladen. Beachten Sie bitte, daß sowohl STX $FB als auch LDX $FB die erläuterte bytesparende Zeropage-Adressierung verwenden.

Ebenso wie bei der Ausgabe des Balls wird der Inhalt des X-Registers über den Akkumulator-Umweg ins Y-Register kopiert und anschließend der Cursor gesetzt, bevor ein Leerzeichen ausgegeben und der Ball damit wieder gelöscht wird.

Das erneute Setzen des Cursors ist nötig, da sich die Cursorposition nach der Ausgabe des Balls veränderte. BSOUT setzt den Cursor nach jedem ausgegebenen Zeichen um eine Spalte nach rechts.

Der erste Schleifendurchgang ist somit beendet. Der Schleifenzähler X wird dekrementiert. Solange Zeile null noch nicht erreicht ist, erfolgt ein weiterer Durchgang (solange DEX nicht zum Ergebnis null führt, ist das Zero-Flag gelöscht und damit die Bedingung "verzweige, wenn ungleich null" erfüllt).

Im zweiten Durchgang besitzt der Schleifenzähler einen um eins verminderten Wert. Der gesamte Vorgang wiederholt sich somit in Spalte 22 von Zeile 22 und so weiter.

Der prinzipielle Ablauf zusammengefaßt:

1. Das Ball-Zeichen wird in der durch den aktuellen X-Wert bestimmten Zeile (und identischer Spalte) ausgegeben.

2. Das Programm wartet auf die Betätigung einer beliebigen Taste.

3. Der zuvor ausgegebene Ball wird durch Überschreiben mit einem Leerzeichen wieder gelöscht, nachdem der Cursor zuvor wieder auf die exakte Ballposition gesetzt wurde.

4. Der Zeilenzähler wird dekrementiert und ein neuer Schleifendurchgang beginnt, außer wenn soeben Zeile 1 bearbeitet wurde.

Die Abbildung einer eigenen C16, Plus-Version des Programms ist überflüssig, da die Unterschiede zum vorgestellten Programm minimal sind. Geben Sie das Programm ab $03F7 ein. Der Schleifenanfang (TXA) besitzt statt $C007 die Adresse $03FE. Ändern Sie entsprechend den Branch-Befehl am Schleifenende von BNE $C007 in BNE $03FE.

### 13.5.2. String-Ausgabe

Dieses Programmprojekt ist sehr anspruchsvoll, liefert Ihnen aber dafür zum Schluß ein allgemein verwendbares Unterprogramm, das Sie in den verschiedensten Programmen benutzen können. Es demonstriert zugleich den effektiven Einsatz von BSOUT und der Unterprogrammtechnik.

Unter einem String verstehen wir eine beliebige Zeichenkette. Bisher war es nur möglich, immer die gleichen Zeichen (A,B,...) auf dem Bildschirm auszugeben.

Bestimmt wünschen Sie sich eine Routine, die wie der PRINT-Befehl beliebige Zeichenketten ausgibt. Wir wissen bereits, wie mit Hilfe der BSOUT-Routine ein Zeichen ausgegeben wird. Der Akkumulator wird mit dem ASCII-Code des Zeichens geladen und BSOUT aufgerufen. Das Zeichen wird an der aktuellen Cursorposition ausgegeben und dieser um eine Stelle weiterbewegt.

Vor der eigentlichen Zeichenausgabe sollte jedoch der Cursor auf die gewünschte Position gesetzt werden. Geben Sie den folgenden Programmteil bitte noch nicht ein. Auf den folgenden Seiten wird das Listing komplett abgebildet.

```
A C000 A2 05      LDX #$05      ;ZEILE 5
A C002 A0 00      LDY #$00      ;SPALTE 0
A C004 18         CLC
A C005 20 F0 FF   JSR $FFF0     ;CURSOR SETZEN
```

Stören Sie sich nicht an dem noch unbekannten Befehl CLC, den die Betriebssystem-Routine zum Setzen des Cursors benötigt. Geben Sie das Programm (ohne Kommentare!) ein, rufen Sie es jedoch noch nicht auf.

Nun folgt der Hauptteil des Programms, die Stringausgabe. Der String muß (siehe BSOUT) in ASCII-Format im Speicher vorliegen. Wir benötigen einen unbenutzten Speicherbereich zur Ablage der einzelnen Bytes. Wir werden den Bereich $033C-$03F6 verwenden, da dieser Bereich sowohl beim C64 als auch beim C16 und Plus/4 nur für Cassettenoperationen (LOAD, SAVE...) verwendet wird.

Die meisten unter Ihnen besitzen wahrscheinlich eine Floppy. Der Bereich $033C-$03F6 ist in diesem Fall immer unbenutzt. Ein Trost für Datasetten-Besitzer: auch Sie können diesen Bereich verwenden. Erst wenn Sie ein Programm laden/speichern, wird die Zeichenkette überschrieben werden.

Das Hauptprogramm soll Zeichen für Zeichen aus diesem Bereich lesen und mit BSOUT ausgeben, wozu wir die indizierte Adressierung verwenden.

```
A C008 A2 00      LDX #$00       ;X INITIALISIEREN
A C00A BD 03 3C   LDA $033C,X    ;ZEICHEN AN ADRESSE $033C + X
A C00D F0 07      BEQ $C016      ;ASCII-CODE = 0 ?
A C00F 20 D2 FF   JSR $FFD2      ;ZEICHEN AUSGEBEN
A C012 E8         INX            ;ZAEHLER INKREMENTIEREN
A C013 4C 0A C0   JMP $C00A      ;NAECHSTES ZEICHEN
A C016 00         BRK            ;ALLES AUSGEGEBEN
```

Der Ablauf dieses doch recht komplexen Programms:

1. Vor Beginn der Schleife ($C00A-$C013) wird das X-Register mit $00 geladen.

2. Der erste Schleifenbefehl (LDA $033C,X) lädt den Akkumulator mit dem Inhalt der Speicherzelle $033C plus X. Das X-Register besitzt beim ersten Durchgang den Inhalt $00, die endgültige Adresse ist daher $033C + $00 = $033C, also der Beginn unseres Strings.

3. Der Akkumulator enthält nun das erste Stringzeichen. BEQ $C016 prüft, ob das Ergebnis der letzten Operation (LDA) gleich null war. Wenn ja, wird zu Adresse $C016 verzweigt.

   Der Sinn dieser Überprüfung: unsere Routine muß in der Lage sein, das Ende eines Strings zu erkennen. Wir markieren dieses Ende, indem dem letzten Zeichen das Byte $00 folgt. Wenn dieses Byte mit LDA $033C,X gelesen wird, verzweigt BEQ zum Programmende.

4. Das im ersten Durchgang gelesene Zeichen wird mit BSOUT ausgegeben (JSR BSOUT).

5. Der Zähler X, der momentan den Inhalt $00 besitzt, wird erhöht (INX) und zeigt damit im nächsten Durchgang auf das zweite Zeichen des Strings ($033C + $01 = $033D).

6. Der Befehl JMP $C00A verzweigt immer (!) zum Schleifenanfang (unbedingter Sprungbefehl; nicht von einer Bedingung abhängig).

7. Im zweiten Durchgang wird der Akkumulator mit dem Inhalt von Adresse $033C + X, also $033D (X=$01) geladen, dem zweiten Zeichen des Strings. Wenn auch dieses Byte ungleich null ist, wird der Branch-Befehl BEQ wiederum nicht ausgeführt und das Zeichen ebenfalls mit BSOUT ausgegeben.

8. Das Programm hangelt sich auf diese Weise durch den gesamten String. X wird nach jedem Durchgang jeweils um eins erhöht und im folgenden Durchgang somit auf die nächste Speicherzelle zugegriffen (siehe indizierte Adressierung).

9. Das Programm ist beendet, wenn mit dem LDA-Befehl das Byte $00 geladen wird, unsere Endemarke, die wir an das Stringende anfügen werden. Da in diesem Fall die letzte Operation (LDA) das Ergebnis null ergab, ist die Bedingung BEQ (ADRESSE) "verzweige, wenn gleich null" erfüllt und der Branch-Befehl verzweigt zum Programmende.

Das Gesamtprogramm (C64):

```
A C000 A2 05      LDX #$05       ;ZEILE 5
A C002 A0 00      LDY #$00       ;SPALTE 0
A C004 18         CLC
A C005 20 F0 FF   JSR $FFF0      ;CURSOR SETZEN
A C008 A2 00      LDX #$00       ;X INITIALISIEREN
A C00A BD 3C 03   LDA $033C,X    ;ZEICHEN AN ADRESSE $033C + X
A C00D F0 07      BEQ $C016      ;ASCII-CODE = 0 ?
A C00F 20 D2 FF   JSR $FFD2      ;ZEICHEN AUSGEBEN
A C012 E8         INX            ;ZAEHLER INKREMENTIEREN
A C013 4C 0A C0   JMP $C00A      ;NAECHSTES ZEICHEN
A C016 00         BRK            ;ALLES AUSGEGEBEN
```

Das Gesamtprogramm (C16, Plus/4):

```
A 03F7 A2 05      LDX #$05      ;ZEILE 5
A 03F9 A0 00      LDY #$00      ;SPALTE 0
A 03FB 18         CLC
A 03FC 20 F0 FF   JSR $FFF0     ;CURSOR SETZEN
A 03FF A2 00      LDX #$00      ;X INITIALISIEREN
A 0401 BD 3C 03   LDA $033C,X   ;ZEICHEN AN ADRESSE $033C + X
A 0404 F0 07      BEQ $040D     ;ASCII-CODE = 0 ?
A 0406 20 D2 FF   JSR $FFD2     ;ZEICHEN AUSGEBEN
A 0409 E8         INX           ;ZAEHLER INKREMENTIEREN
A 040A 4C 01 04   JMP $0401     ;NAECHSTES ZEICHEN
A 040D 00         BRK           ;ALLES AUSGEGEBEN
```

Nach der kompletten Eingabe dieses Programms müssen wir noch eine Zeichenkette ab $033C (=dezimal 828) ablegen. Am einfachsten ist eine BASIC-Schleife, die die ASCII-Codes eines Strings zeichenweise POKEt. Verlassen Sie den Monitor und geben Sie ein:

```
100 A$="DIES IST EIN TEST"
110 FOR I=1 TO LEN(A$):POKE 827+I,ASC(MID$(A$,I,1)):NEXT
120 POKE 827+I,0:REM ENDEMARKE $00
```

Starten Sie das BASIC-Programm und rufen Sie anschließend den Monitor auf. Das Assembler-Programm rufen Sie wie immer mit G C000 (G 03F7) auf.

Wenn Sie das Programm korrekt eingegeben haben, wird auf dem Bildschirm ab Spalte null in Zeile fünf die Zeichenkette DIES IST EIN TEST ausgegeben.

Das Programm können Sie beliebig variieren. Duch Änderung der X- und Y-Werte am Programmanfang bestimmen Sie die Ausgabeposition. Durch Änderung des Strings <A$> im BASIC-Programm kann die auszugebende Zeichenkette geändert werden.

Die Routine ist so allgemein, daß beliebige (!) Zeichenketten ausgegeben werden, wenn folgende Bedingungen beachtet werden:

1. Die Zeichenkette muß ab $033C im Speicher abgelegt werden und darf nicht länger als 187 Zeichen sein ($033C-$03F6).

2. Dem letzten Zeichen muß unbedingt das Byte $00 folgen, da die Routine sonst das Stringende nicht erkennt.

Sollte Sie die umständliche Speicherung der Zeichenkette frustrieren, für die wir ein eigenes BASIC-Programm benötigen: in einem späteren Kapitel werden Programmierhilfsmittel erläutert, die eine komfortablere Eingabe von Assembler-Programmen gestatten als ein Monitor (dennoch ist auch mit diesen Hilfsmitteln ein Monitor-Programm unverzichtbar!).

Bei Verwendung eines solchen Hilfsprogramms werden Texte unmittelbar eingegeben. Das Programm erledigt für Sie die Aufgabe, die einzelnen Textzeichen in ASCII-Form abzulegen.

Ein Tip: mit dem Hauptteil des vorgestellten Programms (ab LDX #$00) verfügen Sie über eine allgemeine Ausgabe-Routine, die Sie in beliebige Programme einbauen können. Geschickterweise bauen Sie die Routine als Unterprogramm ein, das mit RTS (return from subroutine, etwa: verlasse das Unterprogramm) anstatt mit BRK beenden.

Das Unterprogramm kann von beliebigen Programmteilen aus mit JSR (ADRESSE) aufgerufen werden, wie im folgenden Beispiel:

Allgemeine String-Ausgaberoutine (C64):

```
A C000 A2 00      LDX #$00      ;X INITIALISIEREN
A C002 BD 3C 03   LDA $033C,X   ;ZEICHEN AN ADRESSE $033C + X
A C005 F0 07      BEQ $C00E     ;ASCII-CODE = 0 ?
A C007 20 D2 FF   JSR $FFD2     ;ZEICHEN AUSGEBEN
A C00A E8         INX           ;ZAEHLER INKREMENTIEREN
A C00B 4C 02 C0   JMP $C002     ;NAECHSTES ZEICHEN
A C00E 60         RTS           ;RETURN FROM UNTERPROG.

A C00F A9 93      LDA #$93      ;ASCII-CODE $93 (147)
A C011 20 D2 FF   JSR $FFD2     ;AUSGEBEN => CLEAR SCREEN
A C014 A0 0F      LDY #$0F      ;ZAEHLER MIT $0F LADEN
A C016 20 00 C0   JSR $C000     ;AUSGABE-UNTERPROG AUFRUFEN
A C019 A9 0D      LDA #$0D      ;ASCII-CODE $0D (13)
A C01B 20 D2 FF   JSR $FFD2     ;AUSGEBEN => ZEILENVORSCHUB
A C01E 88         DEY           ;ZAEHLER DEKREMENTIEREN
A C01F D0 F5      BNE $C016     ;SCHLEIFE BEENDET?
A C021 00         BRK           ;STRING 16MAL AUSGEGEBEN
```

Zur besseren optischen Unterscheidung befindet sich eine (nicht einzugebende!) Leerzeile zwischen dem Unter- und dem Hauptprogramm.

Das Hauptprogramm erfordert eine eigene Erläuterung. Ziel des Hauptprogramms ist es, zuerst den Bildschirm zu löschen und anschließend mit Hilfe der Ausgaberoutine den vom BASIC-Programm angelegten String <A$> 15mal auszugeben.

Zum Löschen des Bildschirms wird das Steuerzeichen der Taste CLR verwendet (ASCII-Code 147, als Hex-Zahl $93). Wie beschrieben können mit BSOUT beliebige (!) ASCII-Codes ausgegeben werden.

Der erzielte Effekt ist identisch mit dem BASIC-Befehl PRINT CHR$(147). Da die Ausgaberoutine in einer Schleife 16mal aufgerufen wird, lädt LDY #$0F das als Schleifenzähler verwendete Y-Register mit diesem Wert.

Das Unterprogramm befindet sich vor (!) dem Hauptprogramm und wird mit JSR $C000 aufgerufen, der Inhalt von <A$> einmal ausgegeben.

Nun wird mit BSOUT der ASCII-Code $0D (dezimal 13) ausgegeben. Wenn Sie in der ASCII-Tabelle nachschauen, stellen Sie fest, daß der Code 13 der RETURN-Taste zugeordnet ist. Die Ausgabe dieses Codes bewirkt somit einen Zeilenvorschub, der Cursor wird auf die erste Spalte der nächsten Zeile gesetzt.

Das Zählregister Y wird mit DEY vermindert und die Schleife erneut durchlaufen, wenn die Dekrementierung nicht zum Ergebnis null führte. Wenn doch, wird der Branch-Befehl BNE $C016 nicht ausgeführt und das Programm mit BRK beendet.

Geben Sie dieses Programm bitte ein und starten Sie es nicht mit G C000, sondern mit G C00F. Bedenken Sie, das Hauptprogramm beginnt erst hinter dem Ende des Unterprogramms!

Das Programm ist nur in einer C64-Version abgebildet. Für C16, Plus/4-Besitzer ist es eine eine gute Übung, das Programm mit der Startadresse $03F7 einzugeben und die Sprungadressen entsprechend zu ändern. Geben Sie zuerst willkürliche Adressen als Sprungziele der BNE-, JSR- und JMP-Befehle ein. Die Adressen korrigieren Sie nach Eingabe des kompletten Programms anhand der vom Monitor ausgegebenen Adressen der verschiedenen Befehle.

Das Demoprogramm zeigt nicht nur den effektiven Einsatz von BSOUT, sondern auch die Gliederung von Haupt- und Unterprogrammen.

Allgemeine Unterprogramme sollten Sie bei Eingabe mit dem Monitor immer vor (!) dem Hauptprogramm eingeben. Unabhängig vom folgenden Hauptprogramm befinden sich die Unterprogramme dann immer an der gleichen Stelle im Speicher und können immer gleich aufgerufen werden (in diesem Fall mit JSR $C000).

Befindet sich das Hauptprogramm vor dem Unterprogramm, ändert sich die Startadresse des Unterprogramms, je nach Länge des davor liegenden Hauptprogramms. Jedesmal, wenn Sie die Routine in einem Programm verwenden wollen, müssen Sie sich anschauen, ab welcher Adresse das Unterpropramm nun diesmal beginnt.

Bei dem gezeigten Unterprogramm müssen Sie sogar die Sprungadresse des JMP-Befehls, den das Unterprogramm enthält, jedesmal ändern, ein völlig unnötiger Arbeitsaufwand, den Sie sich mit der beschriebenen Methode ersparen.

## 14. Rechnen in Maschinensprache

Das Rechnen in Maschinensprache ist nicht ganz so einfach wie in BASIC, da unser Prozessor nur über Additions- und Subtraktionsbefehle verfügt. Das Programm läßt sich jedoch lösen, indem Multiplikationen und Divisionen auf die grundlegenden Operationen Addition/Subtraktion zurückgeführt werden.

Lassen Sie sich durch diese Einschränkung nicht entmutigen. In den wenigsten Fällen wird in einem Assembler-Programm eine Multiplikation oder Subtraktion benötigt. Wir programmieren ja nicht in Assembler, um eine Zinsberechnung durchzuführen oder ein Buchhaltungsprogramm zu schreiben. Für derartige Probleme ist eine höhere Programmiersprache wie BASIC weitaus besser geeignet.

Mit Assembler wollen wir blitzartig Daten im Speicher hin- und herschaufeln oder vielleicht Sortier- und Eingaberoutinen für BASIC-Programme schreiben (der BASIC-Befehl INPUT ist eine Zumutung). In allen diesen Fällen werden wir die Rechenfähigkeiten von BASIC niemals vermissen.

Doch nun zu den Additions- und Subtraktionsbefehlen, die wir in vielen Programmen benötigen. Arithmetische Operationen finden immer (!) im Akkumulator statt.

Das heißt, wenn wir zwei Zahlen addieren wollen, laden wir die erste Zahl in den Akkumulator und geben anschließend an, welche Zahl zum Akkumulatorinhalt zu addieren ist. Das Ergebnis einer solchen arithmetischen Operation befindet sich anschließend ebenfalls immer im Akkumulator!

ADC (subtract with carry, addiere mit Carry-Bit) ist der Additions-Befehl unseres Rechners, SBC (subtract with carry, subtrahiere mit Carry-Bit) der Subtraktionsbefehl.

Das Carry-Bit oder Carry-Flag ist Bit Nummer 0 des Status-Registers (ganz rechts in der Dualzahl %00000000). Wozu benötigen wir dieses Flag bei arithmetischen Operationen?

Bedenken Sie folgendes: der Akkumulator kann als 8-Bit-Register wie jede andere Speicherzelle auch nur Werte zwischen null und 255 annehmen. Angenommen, wir addieren die Zahlen 255 ($FF oder dual %11111111) und eins ($01 oder dual %00000001). Das Ergebnis 256 ist größer als 255 und muß daher zwangsläufig falsch dargestellt werden.

Um auszuprobieren, was sich in diesem Fall ereignet, geben Sie bitte ein (ab $03F7 beim C16, Plus/4):

```
A C000 A9 FF      LDA #$FF      ;AKKU MIT $FF LADEN
A C002 18         CLC           ;WIRD NOCH ERLÄUTERT
A C003 69 01      ADC #$01      ;$01 ZUM AKKU ADDIEREN
A C005 00         BRK
```

Nach dem Aufruf dieses Programms stellen Sie anhand der Registeranzeige fest, daß der Akkumulator den Wert null ($00) enthält!

Analog zu den Inkrementierbefehlen wird somit nach dem maximalen Wert $FF wieder am entgegengesetzten Ende begonnen, mit dem Wert $00. Wenn Sie den Befehl ADC #$01 durch ADC #$02 ersetzen, wird daher das Ergebnis (im Akkumulator) $02 lauten.

Interessant ist nun, daß im Status-Register das Bit 0 (das Carry-Flag) gesetzt wurde. Die rechte Ziffer des Status-Registers lautet $7, also %0111. Ändern Sie zum Vergleich den Befehl LDA #$FF in LDA #$F0 und rufen Sie das Programm erneut auf.

Die ausgegebene rechte Ziffer des Status-Registers ist nun - nach der Addition von $F0 und $01 (=$F1) - eine 4, also dual %0100, das Carry-Flag ist gelöscht. Am Zustand des Carry-Flags können wir somit erkennen, ob ein Übertrag stattfand und das Ergebnis größer ist als $FF.

Das Carry-Bit wird nicht automatisch gelöscht, wenn das Ergebnis einer Addition $FF nicht überschreitet. Wir müssen daher dafür sorgen, daß es vor der Addition unbedingt gelöscht ist, wenn es uns nicht in die Irre führen soll.

Mit dem Befehl CLC (clear carry, lösche das Carry-Bit) können wir dieses Bit per Hand löschen, um auszuschalten, daß es bereits vor der Addition zufällig (durch irgendeine vorhergehende Operation) noch gesetzt ist. Wenn wir auf diese Weise vorgehen, zeigt uns das Carry-Bit zuverlässig an, ob die Addition zu einem Übertrag führte.

Das Carry-Flag besitzt außer dieser Übertragsanzeige eine weitere Funktion. Es stellt eine Art Verlängerung des Akkumulators dar, gewissermaßen dessen neuntes Bit (Bits 0-8). Das folgende Schema zeigt den Aufbau beider 8-Bit-Register mit dem Carry-Bit als Bit 0 des Status-Registers.

```
Status-Register      Akkumulator

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

Der Sinn des Carry-Bits wird klar, wenn wir eine Addition vornehmen, bei der das Ergebnis größer ist als $FF

```
255    11111111
+ 1   +00000001
--------------
256   100000000
```

Wie Sie sehen, sind alle Bits einer achtstelligen Dualzahl gelöscht. Zur Darstellung dieser Zahl werden neun Bits benötigt. Als dieses neunte Bit fungiert das Carry-Flag (=Carry-Bit), das bei einem solchen Übertrag gesetzt wird. Der Befehl ADC ("addiere mit Carry") addiert nicht nur zwei Zahlen, sondern zählt immer das Carry-Flag hinzu. Wichtig ist diese Arbeitsweise bei der Addition von Zahlen, die größer sind als $FF, also mit zwei Byte dargestellt werden.

Solche Zwei-Byte-Zahlen (oder gleichbedeutend 16-Bit-Zahlen) kennen Sie bereits, nämlich absolut angegebene Adressen ($0400, $C000 etc.). Wie wir wissen, werden diese Adressen in zwei Speicherzellen im Format Low-Byte/High-Byte gespeichert.

Die Zahl $0400 wird z.B. so gespeichert: $00 $04, d.h., in einer Speicherzelle befindet sich das Low-Byte und in der folgenden das High-Byte der Zahl.

Angenommen, in den Speicherzellen $033C-$033F befinden sich zwei solcher 16-Bit-Zahlen, die wir addieren und in den Speicherzellen $0340-$0341 ablegen wollen.

```
Speicherzellen  Inhalt(Low-Byte/High-Byte)

$033C/$033D          $01/$10 (=$1001)
$033E/$033F          $FF/$10 (=$10FF)
```

Prinzipiell werden 16-Bit-Werte addiert, indem zuerst die Low-Bytes und anschließend die High-Bytes addiert werden. Das Schema sieht so aus:

```
1. LADE AKKU MIT DEM INHALT VON $033C ; LOW-BYTE VON ZAHL 1 ($01)
2. ADDIERE DEN INHALT VON $033E       ; LOW-BYTE VON ZAHL 2 ($FF)
3. ERGEBNIS IN $0340 ABLEGEN          ; LOW-ERGEBNIS ($00)

4. LADE AKKU MIT DEM INHALT VON $033D ; HIGH-BYTE VON ZAHL 1 ($10)
5. ADDIERE DEN INHALT VON $033F       ; HIGH-BYTE VON ZAHL 2 ($10)
6. ERGEBNIS IN $0341 ABLEGEN          ; HIGH-ERGEBNIS ($20)
```

Nach Ausführung dieser Operationen enthält die Speicherzelle $0340 das niederwertige Ergebnis $00, die Speicherzelle $0341 das höherwertige Ergebnis $20. Als Gesamtergebnis ergibt sich $2000, das in $0340/$0341 im Low-/High-Format ($00/$20) gespeichert ist.

Das Ergebnis ist offensichtlich falsch ($1001 + $10FF = $2100), da der Übertrag nicht berücksichtigt wurde, der sich bei der Addition der Low-Bytes ergab.

Der Befehl ADC berücksichtigt diesen Übertrag netterweise für uns. Die Addition der Low-Bytes führt zu einem Übertrag und das Carry-Bit wird gesetzt. Bei der folgenden Addition $10 + $10 = $20 wird das Carry-Bit (gesetzt: 1; gelöscht: 0), zum Ergebnis addiert und es ergibt sich korrekterweise der Wert $21.

```
Addition von $1001 und $10FF

A C000 A9 01      LDA #$01      ;LOW-BYTE VON $1001 IN
A C002 8D 3C 03   STA $033C     ;$033C ABLEGEN
A C005 A9 10      LDA #$10      ;HIGH-BYTE VON $1001 IN
A C007 8D 3D 03   STA $033D     ;$033D ABLEGEN
A C00A A9 FF      LDA #$FF      ;LOW-BYTE VON $10FF IN
A C00C 8D 3E 03   STA $033E     ;$033E ABLEGEN
A C00F A9 10      LDA #$10      ;HIGH-BYTE VON $10FF IN
A C011 8D 3F 03   STA $033F     ;$033F ABLEGEN

A C014 18         CLC           ;CARRY-FLAG LOESCHEN !!!
A C015 AD 3C 03   LDA $033C     ;LOW-BYTE VON ZAHL 1
A C018 6D 3E 03   ADC $033E     ;UND ZAHL 2 ADDIEREN
A C01B 8D 40 03   STA $0340     ;ERGEBNIS NACH $0340
A C01E AD 3D 03   LDA $033D     ;HIGH-BYTE VON ZAHL 1
A C021 6D 3F 03   ADC $033F     ;UND ZAHL 2 ADDIEREN
A C024 8D 41 03   STA $0341     ;ERGEBNIS NACH $0341
A C027 00         BRK
```

Das Programm besteht aus zwei Teilen. Im ersten Teil werden die Zahlen $1001 und $10FF im Format Low-Byte/High-Byte in $033C-$033F abgelegt.

Diese 16-Bit-Zahlen werden im zweiten Teil addiert. Unumgänglich ist zuvor der Befehl CLC (clear carry, also: lösche das Carry-Bit), um dafür zu sorgen, daß das Carry-Bit nicht noch als Ergebnis irgendeiner früheren Addition gesetzt ist und das Ergebnis verfälscht.

Zuerst werden die Low-Bytes addiert und das Ergebnis in $0340 abgelegt. Nun werden die High-Bytes addiert und zum Ergebnis der Wert des Carry-Bits (0 oder 1) dazugezählt. Im Beispiel wurde das Carry-Bit durch den Übertrag bei der Addition der Low-Bytes gesetzt, zur folgenden höherwertigen Addition zählt der Prozessor daher noch eine eins dazu.

Der Übertrag ist damit korrigiert und das Ergebnis, das in $0341 abgelegt wird, stimmt. Sie können sich davon überzeugen, indem Sie eingeben:

```
M 0340
```

Der Monitor gibt nach diesem Befehl die Werte der Speicherzellen $0340-$0347 aus. In den ersten beiden Speicherzellen ($0340 und $0341) befindet sich unser Ergebnis $00 und $21, also die Zahl $2100 im Format Low-Byte/High-Byte.

Die Addition von zwei 16-Bit-Zahlen erfolgt somit automatisch völlig korrekt, wenn wir vor den Additionen das Carry-Flag mit dem CLC-Befehl löschen. Merken Sie sich bitte, daß der Befehl ADC zum Akkumulator-Inhalt einen Wert (oder den Inhalt einer Speicherzelle) plus (!) den Wert des Carry-Bit (0 oder 1) addiert.

Ähnlich einfach können zwei 16-Bit-Zahlen subtrahiert werden. Zuerst werden die Low-, dann die High-Bytes mit dem Befehl SBC (subtract with carry) subtrahiert. Einen eventuellen Untertrag bei der Subtraktion der Low-Bytes (Beispiel: $01-$02) berücksichtigt der SBC-Befehl automatisch. Diesmal muß jedoch vor Beginn der Subtraktion das Carry-Flag nicht gelöscht, sondern mit dem Befehl SEC (set carry-flag, setze das Carry-Flag) gesetzt werden.

Merken Sie sich als allgemeine Regel: Vor Additionen wird das Carry-Flag mit dem Befehl CLC gelöscht, vor Subtraktionen mit dem entgegengesetzt wirkendem Befehl SEC gesetzt!

```
Subtraktion von $2001 und $0002

A C000 A9 01      LDA #$01      ;LOW-BYTE VON $2001 IN
A C002 8D 3C 03   STA $033C     ;$033C ABLEGEN
A C005 A9 20      LDA #$20      ;HIGH-BYTE VON $2001 IN
A C007 8D 3D 03   STA $033D     ;$033D ABLEGEN
A C00A A9 02      LDA #$02      ;LOW-BYTE VON $0002 IN
A C00C 8D 3E 03   STA $033E     ;$033E ABLEGEN
A C00F A9 00      LDA #$00      ;HIGH-BYTE VON $0002 IN
A C011 8D 3F 03   STA $033F     ;$033F ABLEGEN

A C014 18         SEC           ;CARRY-FLAG SETZEN !!!
A C015 AD 3C 03   LDA $033C     ;LOW-BYTE VON ZAHL 1
A C018 6D 3E 03   SBC $033E     ;UND ZAHL 2 SUBTRAHIEREN
A C01B 8D 40 03   STA $0340     ;ERGEBNIS NACH $0340
A C01E AD 3D 03   LDA $033D     ;HIGH-BYTE VON ZAHL 1
A C021 6D 3F 03   SBC $033F     ;UND ZAHL 2 SUBTRAHIEREN
A C024 8D 41 03   STA $0341     ;ERGEBNIS NACH $0341
A C027 00         BRK
```

Dieses Programm subtrahiert von $2001 die Zahl $0002. Bei der Subtraktion der Low-Bytes ($01-$02) entsteht ein Untertrag, der automatisch berücksichtigt wird. Mit M 0340 wird Ihnen das korrekte Ergebnis $1FFF angezeigt ($2001 - $0002 = $1FFF).

Eine weitere Anwendung der Rechenoperationen demonstriert das folgende Programm. Es ermittelt die Länge eines BASIC-Programms (in Byte) und gibt sie auf dem Bildschirm aus.

Um dieses Programm zu verstehen, müssen Sie jedoch wissen, daß in der Zeropage die Adressen enthalten sind, an denen ein BASIC-Programm beginnt bzw. endet. Die BASIC-Adressen sind beim C64 in den Speicherzellen $2B/$2C (BASIC-Anfang) bzw. in $2D/$2E (BASIC-Ende) untergebracht (im Adreßformat, also zuerst Low- und dann High-Byte).

Programmlänge ausgeben

```
A C000 38       SEC         ;SUBTRAKTION VORBEREITEN
A C001 A5 2D    LDA $2D     ;SUBTRAHIEREN: LOW-BYTE ENDE
A C003 E5 2B    SBC $2B     ;MINUS LOW-BYTE ANFANG
A C005 AA       TAX         ;ERGEBNIS NACH X BRINGEN
A C006 A5 2E    LDA $2E     ;SUBTRAHIEREN: HIGH-BYTE ENDE
A C008 E5 2C    SBC $2C     ;MINUS HIGH-BYTE ANFANG
A C00A 20 CD BD JSR $BDCD   ;2-BYTE-ZAHL AUSGEBEN
A C00D 00       BRK
```

Wie bei Subtraktionen erforderlich, wird das Carry-Flag gesetzt. Im Anschluß daran wird die Endadresse des Basicprogramms (in $2D/2E) von der Startadresse (in $2B/$2C) subtrahiert.

Zuerst werden die Low-Bytes subtrahiert. Das Ergebnis wird vom Akkumulator in das X-Register übertragen. Anschließend werden die High-Bytes subtrahiert. Danach befindet sich das Low-Byte des Ergebnisses im X-Register und das High-Byte im Akkumulator.

Genau diese Parameter (X-Register: Low-Byte; Akkumulator: High-Byte) verlangt die Routine INTOUT ($BDCD), eine Routine des BASIC-Interpreters, mit der beliebige Integerwerte auf dem Bildschirm an der aktuellen Cursorposition ausgegeben werden.

Das Ergebnis dieses kleinen Programms ist nicht völlig korrekt. Aus Gründen, deren Erläuterung an dieser Stelle zu weit führt, wird nicht die echte Länge des BASIC-Programms ausgegeben, sondern die Länge plus zwei. Im Normalfall stört diese winzige Differenz jedoch kaum. Wenn Sie wollen, können Sie Ihr frisch erworbenes Wissen anwenden und vom ermittelten Ergebnis die Zahl zwei subtrahieren, um ein völlig korrektes Ergebnis zu erhalten.

Da dieses Programm C64-spezifische Adressen verwendet, folgt für alle Besitzer eines C16, Plus/4 oder C128 eine Vergleichstabelle, die ein problemloses Umschreiben des Programms ermöglicht.

| Rechner: | C64 | C16, Plus/4 | C128 |
|---|---|---|---|
| Zeiger auf Startadresse: | $2B/$2C | $2B/$2C | $2D/2E |
| Zeiger auf Endadresse: | $2D/$2E | $2D/$2E | $2F/$30 |
| INTOUT (Integerausgabe): | $BDCD | $A45F | $8E32 |

## 15. Die Vergleichsbefehle

Bei der Schleifenbildung lernten wir die Branch-Befehle BNE, BEQ, BPL und BMI kennen. Diese Befehle verwendeten wir als Kombination aus einer Bedingung (IF..THEN) und einem Sprungbefehl (GOTO).

Der Haken am Einsatz dieser Befehle war jedoch, daß nur festgelegte Bedingungen zulässig waren. BEQ und BNE erkennen, ob der Wert $00 geladen (LDA, LDX, LDY) oder durch In-/Dekrementierung (DEX, DEY, INX, INY) erreicht wurde.

BPL und BMI stellten fest, ob ein Lade- oder In-/Dekrementierbefehl zu einem Ergebnis führt, das größer ist als $80.

Was tun wir jedoch, wenn wir eine Schleife mit dem Startwert eins und dem Endwert zehn benötigen? Wir vergleichen unseren Zähler mit dem gewünschten Endwert.

Die Compare- oder Vergleichs-Befehle CMP, CPX und CPY vergleichen den Inhalt einer angegebenen Speicherzelle (oder eines Registers) mit einem beliebigen Wert. Der Vergleichswert kann unmittelbar (CMP #$0A, CPX #$0A, CPY #$0A) oder absolut (als Inhalt einer Speicherzelle: CMP $0400, CPX $0400, CPY $0400) angegeben werden.

Selbstverständlich ist auch die kürzere (und schnellere) Zeropage-Adressierung möglich (CMP $04). Voraussetzung ist natürlich, daß sich der Vergleichswert in einer Speicherzelle der Zeropage befindet.

Damit sind alle Adressierungsarten der Befehle CPX und CPY genannt. Der Befehl CMP (der den Akkumulatorinhalt mit einem Wert vergleicht) gestattet zusätzlich die indizierte Adressierung (CMP $0400,X = vergleiche den Akku mit dem Inhalt von Speicherzelle $0400 + X). Nähere Angaben zu den möglichen Adressierungsarten der Compare-Befehle finden Sie im Anhang.

Die Vergleichsbefehle werden erst in diesem Kapitel eingeführt, weil das vorige Kapitel die Grundlage zum Verständnis bildet. Bei einem Vergleichsbefehl wird der Vergleichswert automatisch vom Inhalt des Akkumulators (bei CMP) oder einem der Indexregister (bei CPX und CPY) abgezogen.

Der Registerinhalt bleibt jedoch unverändert erhalten! Dieses merkwürdige Verhalten können Sie sich so vorstellen: der Vergleichswert wird abgezogen, anschließend wird der ursprüngliche Registerinhalt jedoch wiederhergestellt.

Die Frage ist, welchen Sinn diese Subtraktion ohne Resultat besitzt. Ein Vergleichsbefehl setzt oder löscht - je nach Ergebnis der Subtraktion - die drei kennengelernten Flags (Zero-Flag, Negativ-Flag, Carry-Flag).

Folgende Ergebnisse werden durch den Zustand der Flags angezeigt:

1. Der Inhalt des Registers ist größer als der Vergleichswert: Carry-Flag gesetzt (1), Negativ- und Zero-Flag gelöscht (0).

2. Der Inhalt des Registers ist genauso groß wie der Vergleichswert: Carry-Flag und Zero-Flag gesetzt (1), Negativ-Flag gelöscht (0).

3. Der Inhalt des Registers ist kleiner als der Vergleichswert: Negativ-Flag gesetzt (1), Carry-Flag und Zero-Flag gelöscht (0).

Ein Beispiel: Der Befehl CMP #$0A vergleicht den Inhalt des Akkumulators mit dem unmittelbar angegebenen Wert $0A. Angenommen, der Akkumulator enthält ebenfalls den Wert $0A. Bei der folgenden Subtraktion wird $0A (der Vergleichswert) von $0A (dem Akkumulator-Inhalt) subtrahiert.

Das Ergebnis ist null und sowohl das Carry- als auch das Zero-Flag ist gesetzt. Mit BEQ ("verzweige, wenn gleich null") können wir in diesem Fall verzweigen.

Beachten Sie bitte, was das Negativ-Flag (und somit die Befehle BPL und BMI) unter größer bzw. kleiner verstehen. Wenn z.B: ein Register den Wert $98 enthält und mit $05 verglichen wird, wird das Negativ-Flag nicht gelöscht, sondern gesetzt!

Warum? Weil $98-$05 (die automatische Subtraktion bei Vergleichsbefehlen) zum Ergebnis $94 führt, einem negativen Ergebnis (positiv=$00-$79, negativ=$80-$FF).

Merken Sie sich daher bitte: BPL verzweigt nicht immer, wenn der Registerinhalt größer ist als der Vergleichswert, sondern nur dann, wenn die Subtraktion REGISTER - VERGLEICHSWERT zu einem positiven Ergebnis führt (Ergebnis kleiner als $80). Gleiches gilt umgekehrt für BMI.

Diese Besonderheit sollte bei der im folgenden erläuterten Schleifenbildung unbedingt beachtet werden.

## 16. Schleifen und Vergleichsbefehle

Vergleichsbefehle können hervorragend zur Schleifenbildung eingesetzt werden. Wir sahen, daß uns die Flags eindeutige Auskünfte darüber liefern, ob ein Registerinhalt größer, gleich oder kleiner als ein Vergleichswert ist.

Um nun abhängig vom Ergebnis bestimmte Operationen auszuführen, benutzen wir die Branch-Befehle, die bekanntlich die Flags testen und je nach Flag-Zustand verzweigen oder nicht.

Im Normalfall wollen wir unterscheiden, ob der Schleifenzähler größer, gleich oder kleiner als der Vergleichswert ist, analog den folgenden BASIC-Schleifen.

Die Funktion der Branch-Befehle BEQ, BNE, BPL und BMI könnte (ohne näher auf die Flags einzugehen) so ausgedrückt werden:

- BEQ: Verzweige, wenn Register gleich Vergleichswert.
- BNE: Verzweige, wenn Register ungleich Vergleichswert.
- BPL: Verzweige, wenn Register größer oder gleich Vergleichswert und die Differenz nicht größer ist als $79.
- BMI: Verzweige, wenn Register kleiner als Vergleichswert und die Differenz nicht größer ist als $79.

Was der Zusatz "und die Differenz nicht größer ist als $79" bei BPL und BMI zu bedeuten hat, wurde im vorigen Abschnitt erläutert.

Mit dieser Kurzbeschreibung können wir problemlos die folgenden BASIC-Schleifen in Assembler schreiben.

Verzweige, solange Schleifenzähler ungleich Vergleichswert (BASIC)

```
100 X=0
110 PRINT "A";
120 X=X+1
130 IF X<>10 THEN GOTO 110
```

Verzweige, solange Schleifenzähler ungleich Vergleichswert (Assembler)

```
A C000 A2 00       LDX #$00      ;ZAEHLER MIT $00 LADEN
A C002 A9 41       LDA #$41      ;ASCII-CODE FUER 'A'
A C004 20 D2 FF    JSR $FFD2     ;'A' AUSGEBEN
A C007 E8          INX           ;ZAEHLER ERHOEHEN
A C008 E0 0A       CPX #$0A      ;ZAEHLER MIT $0A VERGLEICHEN
A C00A D0 F8       BNE $C004     ;WENN UNGLEICH: WEITERMACHEN
A C00C 00          BRK
```

Dieses Programm verwirklicht zum ersten Mal eine Schleife, die aufwärts zählt. Endlich, werden wohl viele sagen. Der Ablauf entspricht der BASIC-Variante 1, die üblicherweise mit einer FOR..NEXT-Schleife verwirklicht wird.

```
100 FOR I=0 TO 10
110 : PRINT"A";
120 NEXT
```

Der Zähler - das X-Register, zu Beginn mit dem Inhalt $00 initialisiert - wird nach jedem Schleifendurchgang inkrementiert und anschließend mit $0A verglichen (CPX #$0A). Das Zero-Flag wird nur dann gesetzt, wenn beide Werte gleich groß sind (und die Subtraktion daher den Wert null ergibt), bei Ungleichheit wird es gelöscht.

Der Befehl BNE testet dieses Flag und kann daher verwendet werden, um zu prüfen, ob das X-Register bereits den Wert $0A enthält. Solange X nicht den Vergleichswert $0A enthält, verzweigt der BNE-Befehl wieder zum Schleifenanfang.

Das nächste A wird ausgegeben und X wieder inkrementiert. Der Vorgang wiederholt sich zehnmal (X= 0,1..,8,9). Beim zehnten Durchgang besitzt X den Wert $0A, BNE stellt Gleichheit zwischen Register und Vergleichswert fest und die Schleife wird verlassen.

```
Verzweige, solange Schleifenzähler größer/gleich Vergleichswert (BASIC)

100 X=100
110 PRINT "A";
120 X=X-1
130 IF X>=10 THEN GOTO 110
```

Verzweige, solange Schleifenzähler größer/gleich Vergleichswert (Assembler)

```
A C000 A2 64      LDX #$64      ;ZAEHLER MIT $64 LADEN
A C002 A9 41      LDA #$41      ;ASCII-CODE FUER 'A'
A C004 20 D2 FF   JSR $FFD2     ;'A' AUSGEBEN
A C007 CA         DEX           ;ZAEHLER VERMINDERN
A C008 E0 0A      CPX #$0A      ;ZAEHLER MIT $0A VERGLEICHEN
A C00A 10 F8      BPL $C004     ;WENN GRÖSSER/GLEICH: WEITERMACHEN
A C00C 00         BRK
```

In diesem Programm wird der Zähler mit $64 (=dezimal 100) geladen und nach jedem Durchgang dekrementiert. CPX #$0A vergleicht das X-Register mit dem Wert $0A, indem vom Registerinhalt $0A subtrahiert werden.

Die Flags werden je nach Ergebnis der Subtraktion gesetzt oder gelöscht und anschließend mit BPL das Negativ-Flag getestet. BPL verzweigt, wenn das Negativ-Flag gelöscht ist. Dies ist der Fall, wenn der Registerinhalt größer oder gleich dem Vergleichswert ist. Erst wenn diese Bedingung nicht mehr erfüllt ist (X kleiner ist als der Vergleichswert $0A), verzweigt BPL nicht mehr.

Bei der Verwendung von BPL und BMI ist große Vorsicht geboten (denken Sie daran, was im letzten Kapitel zu den Begriffen größer/kleiner und deren spezieller Bedeutung für diese Befehle gesagt wurde).

Wenn Sie im vorliegenden Programm den ersten Befehl (LDX #$64) durch LDX #$FF ersetzten und das Programm erneut aufrufen, werden Sie verblüfft feststellen, daß ein einziges A ausgegeben wird.

Nach dem ersten Durchgang wird der Inhalt von X ($FE) mit $0A verglichen. Die Subtraktion ergibt $F4, ein negatives Ergebnis.
Das Negativ-Flag wird gesetzt und BPL verzweigt nicht. Größer bedeutet für BPL, daß das Ergebnis der Subtraktion beim Vergleichsbefehl positiv ist, also kleiner als $80.

Entsprechendes gilt umgekehrt für BMI (das Ergebnis wird als größer gewertet, wenn das Register um mehr als $79 kleiner ist als der Vergleichswert).

BPL und BMI sollten daher nur für kleine Schleifen mit maximal $79 (127) Durchgängen eingesetzt werden, um unangenehme Überraschungen zu vermeiden.

## 16.1. BCS und BCC

Die Vergleichsbefehle setzen oder löschen außer dem Zero- und dem Negativ-Flag auch das Carry-Flag. Es bietet sich daher an, auch dieses Flag zur Schleifenbildung auszunutzen. Uns fehlen jedoch noch die Befehle, die dieses Flag testen und entsprechend verzweigen oder nicht verzweigen.

BCS (branch on carry set, verzweige bei gesetztem Carry-Bit) verzweigt, wenn das Carry-Flag gesetzt ist. Ein Vergleichsbefehl setzt das Carry-Flag immer dann, wenn der Registerinhalt größer als gleich dem Vergleichswert ist. In diesem Fall verzweigt BCS.

BCS wird daher gern als Ersatz für BPL verwendet, um Probleme mit negativen Vergleichsergebnissen zu vermeiden (wenn X minus Vergleichswert größer ist als $79).

BCC (branch on carry clear, verzweige bei gelöschtem Carry-Bit), verzweigt im umgekehrten Fall, wenn das Carry-Flag gelöscht ist, d.h., wenn beim Vergleich (automatische Subtraktion) ein Unterlauf stattfand. Ein solcher Unterlauf findet statt, wenn das Register (d.h. sein Inhalt) kleiner ist als der Vergleichswert.

BCC besitzt nicht die Problematik von BMI (siehe oben) und wird daher eingesetzt, wenn die Differenz zwischen Register und Vergleichswert größer als $79 ist.

Das letzte Beispiel mit BCS anstatt BPL:

```
A C000 A2 64      LDX #$64        ;ZAEHLER MIT $64 LADEN
A C002 A9 41      LDA #$41        ;ASCII-CODE FUER 'A'
A C004 20 D2 FF   JSR $FFD2       ;'A' AUSGEBEN
A C007 CA         DEX             ;ZAEHLER VERMINDERN
A C008 E0 0A      CPX #$0A        ;ZAEHLER MIT $0A VERGLEICHEN
A C00A B0 F8      BCS $C004       ;WENN GRÖSSER/GLEICH: WEITERMACHEN
A C00C 00         BRK
```

Noch einmal, da dies oft falsch gemacht wird: im Gegensatz zu BPL und BMI ist für BCS und BCC uninteressant, ob ein Vergleichsergebnis positiv oder negativ ist. Beide Befehle können immer problemlos eingesetzt werden, auch dann, wenn zu Beginn der Schleife die Differenz zwischen Zähler und Vergleichswert größer ist als $79.

Denken Sie jedoch bitte daran, daß die in Schleifen häufig verwendeten Befehle INX, DEX, INY und DEY das Carry-Flag nicht beeinflussen! Eine Schleife wie

```
          LDX #$0A
(ADRESSE) DEX
          BCS (ADRESSE)
```

ist daher eine Endlosschleife. Wenn wie in diesem Beispiel BCS das Carry-Flag testen soll, muß zuvor ein Vergleich des X-Registers mit $00 stattfinden.

```
          LDX #$0A
(ADRESSE) DEX
          CMP #$00
          BCS (ADRESSE)
```

Von den bisher behandelten Befehlen beeinflussen das Carry-Flag nur ADC, SBC, CLC, SEC und die Vergleichsbefehle CMP, CPX, CPY.

Noch ein wichtiger Hinweis: Angenommen, ADRESSE - also das Sprungziel - befindet sich nicht vor dem Befehl DEX, sondern vor LDX #$0A. Auch die letzte Version bildet in diesem Fall eine Endlosschleife, da das X-Register nicht wirklich herunter-

gezählt, sondern nach jeder Verzweigung wieder von neuem mit dem Ausgangswert $0A geladen wird.

Achten Sie bitte darauf, daß das Ziel einer Verzweigung niemals die Initialisierung, das Laden eines Registers mit einem Startwert, sondern immer der folgende Befehl ist, also der erste Befehl innerhalb (!) der Schleife.

## 16.2. Schleifenbildung: Zusammenfassung

Wie die vorausgegangenen Kapitel zeigten, ist die Schleifenbildung in Assembler sehr komplex. Sie werden viel Übung benötigen, bis Sie für jeden Fall die optimalen Schleife bilden können.

In diesem Abschnitt will ich weniger über Flags sprechen, sondern Ihnen Faustregeln für die Anwendung der Vergleichs- und Sprungbefehle geben.

Um Ihnen den Umgang mit den Sprungbefehlen zu erleichtern, beschreibe ich deren Funktion noch einmal im wenigen Worten. Beachten Sie bitte unbedingt die Besonderheiten bei BPL und BMI:

-BEQ: Verzweige, wenn Register gleich Vergleichswert.
-BNE: Verzweige, wenn Register ungleich Vergleichs wert.
-BCS: Verzweige, wenn Register größer oder gleich dem Vergleichswert.
-BCC: Verzweige, wenn Register kleiner als Ver gleichswert.
-BPL: Verzweige, wenn Register größer oder gleich Ver gleichswert und die Differenz nicht größer ist als $79.
-BMI: Verzweige, wenn Register kleiner als Vergleichswert und die Differenz nicht größer als $79.

Merken Sie sich bitte vor allem, daß bei Schleifen unbedingt zu berücksichtigen ist, ob die letzte Operation vor einem Branch-Befehl auch tatsächlich das vom betreffenden Befehl getestete Flag beeinflußt.

Folgende bisher besprochene Befehle beeinflussen die uns bekannten Flags (Zero-Flag, Negativ-Flag, Carry-Flag):

| Befehl | Beeinflußte Flags |
|---|---|
| LDA | Negativ, Zero |
| LDX | Negativ, Zero |
| LDY | Negativ, Zero |
| TAX | Negativ, Zero |
| TAY | Negativ, Zero |
| TXA | Negativ, Zero |
| TYA | Negativ, Zero |
| INX | Negativ, Zero |
| INY | Negativ, Zero |
| INC | Negativ, Zero |
| DEX | Negativ, Zero |
| DEY | Negativ, Zero |
| DEC | Negativ, Zero |
| ADC | Negativ, Zero, Carry |
| SBC | Negativ, Zero, Carry |
| CMP | Negativ, Zero, Carry |

Denken Sie bitte daran: alle Branch-Befehle testen den aktuellen (!) Zustand eines Flags und prüfen somit das Ergebnis der letzten (!) Operation, die das betreffende Flag beeinflußte. Was passiert, wenn diese Tatsache nicht berücksichtigt wird, zeigt das folgende Beispiel:

```
          LDX #$0A
(ADRESSE) ...
          ...
          DEX
          LDA #$FF
          BNE $(ADRESSE)
```

In diesem Beispiel wird nach der Dekrementierung des Zählers der Akkumulator mit $FF geladen. BNE testet das Zero-Flag und verzweigt, wenn es gelöscht ist. Die Schleife soll (!) beendet werden, wenn das X-Register den Wert null annimmt und daher das Zero-Flag gesetzt wird.

Der folgende LDA-Befehl beeinflußt das Zero-Flag jedoch ebenfalls. Da $FF ungleich null ist, wird das Zero-Flag unmittelbar vor dem Test mit BNE gelöscht und BNE verzweigt immer (Endlosschleife)!

Abhilfe schafft ein zusätzlicher Vergleichsbefehl, der das X-Register mit $00 vergleicht und nun als letzter (!) Befehl vor BNE das Zero-Flag beeinflußt.

```
          LDX #$0A
(ADRESSE) ...
          ...
          DEX
          LDA #$FF
          CPX #$00
          BNE $(ADRESSE)
```

Zugegeben, dieses Beispiel ist nicht allzu sinnvoll. Im Normalfall wird man den automatischen Vergleich mit $00 vorziehen, der bei DEX erfolgt und eine andere Reihenfolge der Befehle wählen (LDA #$FF : DEX : BNE$(ADRESSE)), durch die sich der Vergleichsbefehl erübrigt.

In der Praxis können jedoch sehr wohl Fälle auftreten, in denen tatsächlich ein expliziter Vergleich des Zählregisters mit $00 angebracht ist, weil nach der Dekrementierung des Zählregisters Befehle folgen, die das getestete Flag beeinflussen und daher das Ergebnis der Dekrementierung verfälschen.

Denken Sie bitte daran, daß bei der Schleifenbildung mit BCC und BCS ein vorangehender Vergleichsbefehl notwendig ist, da die In-/Dekrementierbefehle das von BCC und BCS getestete Carry-Flag nicht beeinflussen.

## 16.3. Übungsprogramme zur Schleifenbildung

Sie kennen nun fast alle Branch-Befehle und können theoretisch bereits Schleifen beliebiger Art bilden. Der richtige Umgang mit den Branch-Befehlen setzt jedoch viel Übung voraus. Daher wird sich dieses Kapitel nicht mit weiteren Befehlen, sondern ausschließlich mit Übungsprogrammen beschäftigen.

### 16.3.1. Steuerung per Tastatur

Mit den Vergleichs- und den Branch-Befehlen sind wir in der Lage, ein Programm zu schreiben, das das Grundelement aller Videospiele demonstriert, die Steuerung eines Objektes durch den Benutzer.

Als Objekt verwenden wir einen Ball, der mit den Cursortasten in beliebige Richtungen bewegt wird. Das Prinzip kann jedoch ebensogut auf Sprites oder Shapes übertragen werden.

Der prinzipielle Ablauf:

1. Ein Zeilenzähler (X-Register) und ein Spaltenzähler (Y-Register) wird ständig die aktuelle Ballposition festhalten.

2. Ein vorbereitender Programmteil löscht den Bildschirm und setzt das X-Register auf den Wert zwölf (Zeile zwölf) und das Y-Register auf den Wert 20 (Spalte 20). Die dadurch festgelegte Position soll die Ausgangsposition des Balls sein.

3. In der nun beginnenden Schleife wird der Ball an der durch X und Y festgelegten Position gezeichnet.

4. Nach der Ausgabe des Balls wird die Tastatur abgefragt. Wurde eine Taste betätigt, wird der Ball an der aktuellen Position gelöscht.

5. Je nach betätigter Cursortaste wird der Zeilenzähler (X-Register) oder der Spaltenzähler (Y-Register) entsprechend verändert.

6. Mit einem absoluten Sprungbefehl (JMP) wird immer zum Schleifenbeginn zurückgekehrt und der Ball an der neuen Position ausgegeben (siehe 3.).

```
Vorbereitung
A C000 A9 93          LDA #$93        ;ASCII-CODE FÜR CLEAR
A C002 20 D2 FF       JSR $FFD2       ;SCREEN AUSGEBEN
A C005 A2 0C          LDX #$0C        ;ZEILENZÄHLER: $0C (=12)
A C007 A0 14          LDY #$14        ;SPALTENZÄHLER: $14 (=20)
Ball an aktueller Position ausgeben
A C009 18             CLC             ;VORBEREITUNG F.PLOT-ROUTINE
A C00A 20 F0 FF       JSR $FFF0       ;PLOT => CURSOR SETZEN
A C00D A9 71          LDA #$71        ;ASCII-CODE FÜR BALLZEICHEN
A C00F 20 D2 FF       JSR $FFD2       ;BSOUT => BALL AUSGEBEN
Tastatur abfragen
A C012 8E 3C 03       STX $033C       ;X-WERT 'RETTEN'
A C015 8C 3D 03       STY $033D       ;Y-WERT 'RETTEN'
A C018 20 E4 FF       JSR $FFE4       ;GETIN => TASTATUR ABFRAGEN
A C01B F0 FB          BEQ $C018       ;TASTE GEDRÜCKT?
A C01D 8D 3E 03       STA $033E       ;TASTE: ZEICHENCODE 'RETTEN'
Ball an aktueller Position löschen
A C020 AE 3C 03       LDX $033C       ;X-WERT ZURÜCKHOLEN
A C023 AC 3D 03       LDY $033D       ;Y-WERT ZURÜCKHOLEN
A C026 18             CLC             ;VORBEREITUNG F.PLOT-ROUTINE
A C027 20 F0 FF       JSR $FFF0       ;PLOT => CURSOR SETZEN
A C02A A9 20          LDA #$20        ;ASCII-CODE FÜR LEERZEICHEN
A C02C 20 D2 FF       JSR $FFD2       ;BSOUT => BALL LÖSCHEN
Spalten- und Zeilenzähler korrigieren
A C02F AD 3E 03       LDA $033E       ;TASTE: ZEICHENCODE HOLEN
A C032 C9 11          CMP #$11        ;CODE FÜR 'CURSOR DOWN'?
A C034 D0 04          BNE $C03A       ;SPRUNG, WENN NEIN
A C036 E8             INX             ;SONST ZEILE ERHÖHEN
A C037 4C 09 C0       JMP $C009       ;UND ZUM SCHLEIFENANFANG
A C03A C9 1D          CMP #$1D        ;CODE FÜR 'CURSOR RIGHT'?
A C03C D0 04          BNE $C042       ;SPRUNG, WENN NEIN
A C03E C8             INY             ;SONST SPALTE ERHÖHEN
```

```
A C03F 4C 09 C0     JMP $C009       ;UND ZUM SCHLEIFENANFANG
A C042 C9 91        CMP #$91        ;CODE FÜR 'CURSOR UP'?
A C044 D0 04        BNE $C04A       ;SPRUNG, WENN NEIN
A C046 CA           DEX             ;SONST ZEILE VERMINDERN
A C047 4C 09 C0     JMP $C009       ;UND ZUM SCHLEIFENANFANG
A C04A C9 9D        CMP #$9D        ;CODE FÜR 'CURSOR LEFT'?
A C04C D0 BB        BNE $C009       ;SPRUNG, WENN NEIN
A C04E 88           DEY             ;SONST SPALTE VERMINDERN
A C04F 4C 09 C0     JMP $C009       ;UND ZUM SCHLEIFENANFANG
```

Um Ihnen den Überblick zu erleichtern, wurde das Listing entsprechend dem prinzipiellen Ablauf unterteilt und mit (nicht abzutippenden) Kommentaren versehen.

Der vorbereitende Teil löscht den Bildschirm, indem der ASCII-Code der Taste CLR (147 = $93) ausgegeben wird. Anschließend werden die Register mit jenen Werten initialisiert, die der Bildschirmmitte entsprechen. Denken Sie beim folgenden Ablauf immer daran, daß das X-Register die Zeile und das Y-Register die Spalte bestimmen soll.

Teil 2 bildet den Schleifenanfang. Der Cursor wird mit der PLOT-Routine des Betriebssystems auf die durch X und Y festgelegte Bildschirmposition gesetzt. Anschließend wird an der jeweiligen Position mit BSOUT das Ballzeichen (ASCII-Code $71 = 113) ausgegeben.

Der folgende Teil enthält eine Warteschleife, die mit der Routine GETIN auf die Betätigung einer Taste wartet. Da GETIN alle drei Register verändert, müssen zuvor die aktuellen X- und Y-Werte gerettet werden. Die Inhalte dieser Register werden in die Speicherzellen $033C und $033D kopiert. Nachdem die Warteschleife verlassen wird, enthält der Akkumulator den ASCII-Code der gedrückten Taste. Da der Akkumulatorinhalt vom folgenden Programmteil verändert wird, muß dieser ASCII-Code ebenfalls gerettet werden (in $033E).

Der Cursor wird nun wieder mit PLOT auf die aktuelle Ballposition gesetzt (nachdem zuvor die ursprünglichen Werte der Indexregister aus den Speicherzellen zurückgeholt wurden) und der

Ball durch Überschreiben mit einem Leerzeichen (ASCII-Code $20 = 32) gelöscht.

Den letzten Teil des Programms bildet die Veränderung der Indexregister je nach betätigter Cursortaste. Dieser Programmteil zeigt, wie mit Hilfe der Vergleichsbefehle ein Register mit beliebigen Werten verglichen wird.

Der ASCII-Code der betätigten Taste wurde in $033E zwischengespeichert (STA $033E). Dieser Code wird nun zurückgeholt (LDA $033E) und mit den ASCII-Codes der vier Cursortasten (DOWN, RIGHT, UP, LEFT) verglichen.

Der Befehl CMP #$11 vergleicht den Inhalt des Akkumulators mit dem Wert $11, dem ASCII-Code der Taste CURSOR DOWN. Bei Ungleichheit verzweigt BNE nach $C03A, ansonsten wird der folgende Befehl INX ausgeführt und der Zeilenzähler erhöht. Nach dieser Korrektur der Ballzeile erfolgt ein Sprung zum Schleifenanfang (JMP $C009), wo der Ball an der neu festgelegten Position ausgegeben wird.

Die folgenden Abschnitte dieses Programmteils verlaufen analog. Der Akkumulator wird mit dem ASCII-Code einer Cursortaste verglichen. Bei Ungleichheit wird zum nächsten Abschnitt verzweigt. Findet keine Verzweigung statt, wurde die mit CMP überprüfte Taste betätigt und die durch X und Y festgelegte Ballposition entsprechend korrigiert.

Nach dem Aufruf dieses Programms können Sie den Ball mit den vier Cursortasten in beliebige Richtungen steuern. C16- und Plus-Besitzer geben das Program wie gewohnt ab $03F7 ein und ändern die Branch- und JMP-Befehle entsprechend ab.

Achten Sie bitte darauf, den Ball nicht aus dem Bildschirm heraus zu bewegen, da sonst unkontrolliert Speicherzellen außerhalb des Bildschirmspeichers verändert werden (und dies kann zu merkwürdigen Resultaten führen).

Ich stelle Ihnen nun ein BASIC-Programm vor, dessen Ablauf exakt dem Assembler-Programm entspricht. Dessen Ablauf kann durch Vergleich mit den entsprechenden Abschnitten des BASIC-Programms sehr gut nachvollzogen werden.

```
100 PRINT CHR$(147):REM BILDSCHIRM LOESCHEN
110 X=12:Y=20:REM POSITION INITIALISIEREN
120 :
130 POKE 214,X:POKE 211,Y:SYS 58732:REM CURSOR SETZEN
140 PRINT CHR$(113);:REM BALL AUSGEBEN
150 :
160 GET A$:IF A$="" THEN 160
170 A=ASC(A$):REM ASCII-CODE DER TASTE
180 :
190 POKE 214,X:POKE 211,Y:SYS 58732:REM CURSOR SETZEN
200 PRINT CHR$(32);:REM BALL LOESCHEN
210 :
220 IF A<>17 THEN 250:REM UNGLEICH CURSOR DOWN?
230 X=X+1:REM ZEILE ERHOEHEN
240 GOTO 130:REM ZUM SCHLEIFENANFANG
250 IF A<>29 THEN 280:REM UNGLEICH CURSOR RIGHT?
260 Y=Y+1:REM SPALTE ERHOEHEN
270 GOTO 130:REM ZUM SCHLEIFENANFANG
280 IF A<>145 THEN 310:REM UNGLEICH CURSOR UP?
290 X=X-1:REM ZEILE VERMINDERN
300 GOTO 130:REM ZUM SCHLEIFENANFANG
310 IF A<>157 THEN 130:REM UNGLEICH CURSOR LEFT?
320 Y=Y-1:REM SPALTE VERMINDERN
330 GOTO 130:REM ZUM SCHLEIFENANFANG
```

Eine Erläuterung ist zum Verständnis des BASIC-Programms notwendig: die Befehle POKE 214,X:POKE 211,Y:SYS 58732 sind maschinennah und nur für den C64 gedacht, nicht für den C16 oder Plus/4!

Beim C64 kann der Cursor direkt auf eine bestimmte Bildschirmposition gesetzt werden, wenn in Speicherzelle 214 die gewünschte Zeile und in 211 die Spalte gePOKEt, und anschließend eine Betriebssystem-Routine an Adresse 58732 auf-

gerufen wird. Beim C16, Plus ersetzen Sie diese Befehle bitte durch CHAR 1,Y,X,"".

Sollten Sie sich wundern, daß zum Setzen des Cursors mit der PLOT-Routine das Carry-Flag mit CLC gelöscht werden muß, beim Setzen von BASIC aus jedoch die Übergabe von Spalte und Zeile völlig ausreicht: PLOT besitzt eigentlich zwei Funktionen, das Setzen des Cursors und das Lesen der aktuellen Cursorposition. Welche Funktion Sie wählen, bestimmen Sie mit Hilfe des Carry-Flags.

Wenn dieses Flag vor dem Aufruf mit JSR $FFF0 nicht gelöscht, sondern mit SEC gesetzt wird, wird die momentane Cursorposition von PLOT ermittelt und in den Registern übergeben. Nach dem Aufruf enthält das Y-Register die Spalte und das X-Register die Zeile.

### 16.3.2. Verbesserung des Steuerungsprogramms

Das vorgestellte Programm eignet sich hervorragend, um Ihnen einige Feinheiten der Assembler-Programmierung zu zeigen.

Ein echter Schwachpunkt des Programms ist die fehlende Bereichsüberprüfung. Ein Beispiel: der Ball befindet sich in der obersten Zeile und der Benutzer betätigt CURSOR UP, um ihn eine Zeile noch oben zu bewegen. Da dieser Spezialfall nicht abgefragt wird, wird der Zeilenzähler X dekrementiert und enthält anschließend den Wert $FF ($00 minus $00 => $FF), eine ein wenig hohe Zeilennummer.

Das Problem wird am einfachsten gelöst, indem überprüft wird, ob durch die Neupositionierung die Bildschirmgrenzen überschritten würden. Wenn ja, wird ohne Positionsänderung sofort zum Schleifenanfang zurückgekehrt.

Der Programmteil zur Korrektur der Ballposition wird wie folgt erweitert:

```
Spalten- und Zeilenzähler korrigieren
A C02F AD 3E 03     LDA $033E       ;TASTE: ZEICHENCODE HOLEN
A C032 C9 11        CMP #$11        ;CODE FÜR 'CURSOR DOWN'?
A C034 D0 08        BNE $C03E       ;SPRUNG, WENN NEIN
A C036 E0 17        CPX #$17        ;X MIT $17(=23) VERGLEICHEN
A C038 B0 CF        BCS $C009       ;SPRUNG, WENN GRÖSSER/GLEICH
A C03A E8           INX             ;SONST ZEILE ERHÖHEN
A C03B 4C 09 C0     JMP $C009       ;UND ZUM SCHLEIFENANFANG
A C03E C9 1D        CMP #$1D        ;CODE FÜR 'CURSOR RIGHT'?
A C040 D0 08        BNE $C04A       ;SPRUNG, WENN NEIN
A C042 C0 26        CPY #$26        ;Y MIT $26(=38) VERGLEICHEN
A C044 B0 C3        BCS $C009       ;SPRUNG, WENN GRÖSSER/GLEICH
A C046 C8           INY             ;SONST SPALTE ERHÖHEN
A C047 4C 09 C0     JMP $C009       ;UND ZUM SCHLEIFENANFANG
A C04A C9 91        CMP #$91        ;CODE FÜR 'CURSOR UP'?
A C04C D0 08        BNE $C056       ;SPRUNG, WENN NEIN
A C04E E0 00        CPX #$00        ;X MIT $00 VERGLEICHEN
A C050 F0 B7        BEQ $C009       ;SPRUNG, WENN GLEICH
A C052 CA           DEX             ;SONST ZEILE VERMINDERN
A C053 4C 09 C0     JMP $C009       ;UND ZUM SCHLEIFENANFANG
A C056 C9 9D        CMP #$9D        ;CODE FÜR 'CURSOR LEFT'?
A C058 D0 AF        BNE $C009       ;SPRUNG, WENN NEIN
A C05A C0 00        CPY #$00        ;Y MIT $00 VERGLEICHEN
A C05C F0 AB        BEQ $C009       ;SPRUNG, WENN GLEICH
A C05E 88           DEY             ;SONST SPALTE VERMINDERN
A C05F 4C 09 C0     JMP $C009       ;UND ZUM SCHLEIFENANFANG
```

Bis $C034 bleibt das Programm unverändert. Zur Eingabe des korrigierten Teils geben Sie daher dem Assemble-Befehl als Startadresse $C034 an (A C034 BNE $C03E), bzw. die entsprechende Adresse dieses Befehls beim C16, Plus/4.

Am Beispiel des Programmteils $C032-$C03B (Funktion BALL NACH UNTEN) können die Änderungen stellvertretend für die folgenden Programmteile erläutert werden.

Bevor der Zeilenzähler X erhöht wird, erfolgt eine Überprüfung, ob der untere Bildschirmrand bereits erreicht wurde. Der Befehl CPX #$17 vergleicht den Inhalt des X-Registers mit dem Wert $17 (=dezimal 23) und setzt die Flags entsprechend.

BCS testet das Carry-Flag und verzweigt, wenn der Vergleich ergab, daß der X-Wert größer oder gleich $17 ist, d.h., wenn der untere Rand bereits erreicht ist. Ist X kleiner als $17, verzweigt BCS nicht und der Zeilenzähler wird wie in der letzten Programmversion inkrementiert.

Entsprechendes gilt für die folgenden Teile. Der Spaltenzähler Y wird bei CURSOR RIGHT nur dann inkrementiert, wenn der Vergleich CPY #$26 (=dezimal 38) ergab, daß Y kleiner ist als $26. Wenn Y dagegen gleich oder größer als $26 ist, verzweigt BCS zum Programmanfang.

Da es nicht möglich ist, daß der Zeilenzähler X den Wert $17 überschreitet bzw. Y größer wird als $26, kann im vorliegenden Programm BCS durch BEQ ersetzt werden.

Ein Beispiel hierzu: angenommen, der Benutzer gibt Dauerfeuer mit CURSOR RIGHT. Das Y-Register wird solange inkrementiert, bis der Wert $26 erreicht ist. Besitzt Y diesen Wert, ergibt der Befehl CMP #$26 Gleichheit und BEQ verzweigt ebenso wie zuvor BCS (größer oder gleich) zum Schleifenanfang.

Die Funktionen CURSOR UP und CURSOR LEFT vergleichen das X- bzw. Y-Register mit $00. Ist X gleich $00 (oberste Bildschirmzeile), wird verzweigt (BEQ verzweigt bei Gleichheit) und der Zeilenzähler nicht dekrementiert. Ebenso wird verzweigt, wenn der Spaltenzähler Y gleich $00 ist.

### 16.3.3. Blinkende Bildschirmzeile

Das folgende Demoprogramm dient zur Auffrischung Ihres Wissens über die indirekte Adressierung. Wir kennen alle Befehle und Adressierungsarten, die wir benötigen, um eine Bildschirmzeile blinken zu lassen.

Blinken bedeutet: die betreffende Zeile muß abwechselnd invertiert und wieder normalisiert werden. Da wir im folgenden Programm nicht mit BSOUT arbeiten werden, sondern mit LDA

und STA direkt auf den Bildschirmspeicher zugreifen, schauen Sie sich bitte die Bildschirmcode-Tabelle in Ihrem Handbuch an.

Am Ende der Tabelle befindet sich eine kleine Notiz: "Die Codes 128 bis 255 bilden die reversen Zeichen zu den Codes 0 bis 127". Das heißt, indem wir zu einem bestimmten Bildschirmcode 128 ($80) addieren, erhalten wir die inverse Darstellung des betreffenden Zeichens.

```
Bildschirmcode   Zeichen

      1            A
      2            B
...
...
    129            A invers
    130            B invers
...
...
```

Mit der indizierten Adressierung können wir sehr elegant der Reihe nach auf die 40 Zeichen einer Zeile zugreifen, diese lesen, $80 addieren und wieder (invers) zurückschreiben. Das Programm wird in mehreren Schritten entwickelt.

Version 1

```
A C000 A2 00       LDX #$00        ;ZÄHLER INITIALISIEREN
A C002 BD 00 04    LDA $0400,X     ;$0400 + X NACH AKKU
A C005 18          CLC             ;ADDITION VORBEREITEN
A C006 69 80       ADC #$80        ;$80 ADDIEREN
A C008 9D 00 04    STA $0400,X     ;AKKU NACH $0400 + X
A C00B E8          INX             ;ZÄHLER ERHÖHEN
A C00C E0 28       CPX #$28        ;X MIT $28 (40) VERGLEICHEN
A C00E D0 F2       BNE $C002       ;SPRUNG, WENN UNGLEICH
A C010 00          BRK             ;PROGRAMMENDE
```

In diesem Programm wird eine aufwärts zählende Schleife benutzt. Im ersten Durchgang besitzt X den Inhalt $00. In den Ak-

kumulator wird daher der Bildschirmcode an Adresse $0400 + X = $0400 geladen.

Mit CLC wird die Addition vorbereitet, $80 (=dezimal 128) addiert und der invertierte Bildschirmcode nach $0400 zurückgeschrieben.

Der Zähler wird inkrementiert und mit $28 (=dezimal 40) verglichen. Wurde die Zeile noch nicht komplett bearbeitet, ist X kleiner also $28 und BNE verzweigt, da Ungleichheit festgestellt wird (statt BNE könnte selbstverständlich ebenso BCC ("verzweige, wenn kleiner") verwendet werden.

Im nächsten Durchgang wiederholt sich der Vorgang. Da X jedoch nun den Wert $01 besitzt, wird auf $0401 ($0400 + X) zugegriffen.

C16, Plus/4-Besitzer ändern bitte wieder die Anfangsadresse des Bildschirmspeichers $0400 in $0C00.

Bereits früher erwähnte ich, daß in Assembler-Programmen meist abwärts zählende Schleifen verwendet werden. Warum dies so ist, zeigt das vorliegende Programm. Ob die Bildschirmzeile von rechts nach links (aufwärts zählend) oder aber von links nach rechts (abwärts zählend) invertiert wird, ergibt für die praktische Anwendung sicher keinen Unterschied. Mit einer abwärts zählenden Schleife ist das Programm jedoch kürzer (und eleganter):

Version 2

```
A C000 A2 27      LDX #$27      ;ZÄHLER INITIALISIEREN
A C002 BD 00 04   LDA $0400,X   ;AKKU MIT $0400 + X LADEN
A C005 18         CLC           ;ADDITION VORBEREITEN
A C006 69 80      ADC #$80      ;$80 ADDIEREN
A C008 9D 00 04   STA $0400,X   ;AKKU NACH $0400 + X
A C00B CA         DEX           ;ZÄHLER VERMINDERN
A C00C 10 F4      BPL $C002     ;SPRUNG, WENN POSITIV
A C00E 00         BRK           ;PROGRAMMENDE
```

In dieser abwärts zählenden Schleife wird X mit $27 initialisiert und daher zuerst auf das letzte Zeichen der Zeile zugegriffen ($0400 + $27 = $0427). Bei jedem Durchgang wird der Zähler dekrementiert, das Programm bearbeitet die Zeile beginnend mit Adresse $0427 und arbeitet sich bis zu Adresse $0400 zurück.

Zur Überprüfung, ob die letzte Speicherzelle $0400 bereits behandelt wurde, muß (!) BPL eingesetzt werden. BPL verzweigt, wenn das Ergebnis der letzten Operation positiv war (zwischen $00 und $79). Die letzte Operation besteht aus der Dekrementierung des Zählregisters, das vom Startwert $27 heruntergezählt wird.

Auch beim Wert $00 soll noch einmal verzweigt werden, um im folgenden Durchgang die Speicherzelle $0400 zu behandeln. Zu diesem Zweck ist BPL hervorragend geeignet, da auch das Ergebnis $00 noch positiv ist und erst im folgenden Durchgang ein negatives Ergebnis ($FF) erzielt wird.

Wenn Sie BPL durch BNE ersetzen, werden Sie feststellen, daß die Speicherzelle $0400 nicht mehr invertiert wird. Wenn das X-Register den Wert $01 enthält und nun zu $00 dekrementiert wird, verzweigt BNE ("verzweige, wenn nicht (!) gleich null") nicht mehr.

Sie sehen, für Schleifen, die auch beim Zählerwert null ein letztes Mal durchlaufen werden sollen, ist BPL optimal geeignet. Voraussetzung: die Schleife soll nicht mehr als 127mal durchlaufen werden.

Ist diese Voraussetzung nicht erfüllt, wird ein zusätzlicher Vergleichsbefehl benötigt und BNE oder BCS eingesetzt.

```
          LDX #$(STARTWERT)          LDX #$(STARTWERT)
(ADRESSE) ...              (ADRESSE) ...
          ...                        ...
          DEX                        DEX
          CMP #$00                   CMP #$FF
          BCS $(ADRESSE)             BCS (ADRESSE)
```

Sehen Sie sich diese verschiedenen Versionen bitte genau an. Bei der Programmierung in Assembler ist es außerordentlich wichtig, den Unterschied zwischen den verschiedenen Branch-Befehlen zu kennen, um diese optimal einsetzen zu können.

Kommen wir zum zweiten Programmteil, der Normalisierung der Zeile, die analog verläuft, abgesehen davon, daß nicht $80 addiert, sondern subtrahiert wird.

```
A C000 A2 27      LDX #$27       ;ZÄHLER INITIALISIEREN
A C002 BD 00 04   LDA $0400,X    ;AKKU MIT $0400 + X LADEN
A C005 38         SEC            ;SUBTRAKTION VORBEREITEN
A C006 E9 80      SBC #$80       ;$80 SUBTRAHIEREN
A C008 9D 00 04   STA $0400,X    ;AKKU NACH $0400 + X
A C00B CA         DEX            ;ZÄHLER VERMINDERN
A C00C 10 F4      BPL $C002      ;SPRUNG, WENN POSITIV
A C00E 00         BRK            ;PROGRAMMENDE
```

Beachten Sie bitte, daß das Carry-Bit gesetzt (!) werden muß, um die Subtraktion vorzubereiten.

Es bietet sich nun an, beide Programmteile nicht einfach hintereinanderzuhängen, sondern ein wenig eleganter miteinander zu verbinden. Eine Möglichkeit besteht darin, nur eine Schleife zu verwenden und zu prüfen, ob eine Addition oder aber eine Subtraktion vorzunehmen ist.

Version 3

```
A C000 A2 27      LDX #$27       ;X INITIALISIEREN
A C002 BD 00 04   LDA $0400,X    ;$0400 + X IN AKKU
A C005 30 06      BMI $C00D      ;SPRUNG, WENN NEGATIV
A C007 18         CLC            ;SONST ADD.VORBEREITEN
A C008 69 80      ADC #$80       ;UND $80 ADDIEREN
A C00A 4C 10 C0   JMP $C010      ;SUBTRAKTION ÜBERSPRINGEN
A C00D 38         SEC            ;SUBTRAKTION VORBEREITEN
```

```
A C00E E9 80      SBC #$80       ;$80 SUBTRAHIEREN
A C010 9D 00 04   STA $0400,X    ;AKKU NACH $0400 + X
A C013 CA         DEX            ;ZÄHLER VERMINDERN
A C014 10 EC      BPL $C002      ;SPRUNG, WENN POSITIV
A C016 4C 00 C0   JMP $C000      ;NEUSTART DES PROGRAMMS
```

Erstaunlicherweise ist dieses Programm nur unwesentlich länger als einer der beiden Teile Zeile invertieren und Zeile normalisieren.

Der Grund liegt im äußerst effektiven Einsatz von BPL. Nach dem Laden des Inhaltes der Speicherzelle $0400,X prüft BPL, ob der geladene Wert positiv oder negativ ist.

Ist er negativ (größer als $79), verzweigt BMI ("verzweige, wenn negativ") zum Teil "$80 subtrahieren und das offensichtlich bereits invertierte Zeichen wird normalisiert.

Ist der geladene Wert dagegen positiv, also kleiner oder gleich $79, ist das Zeichen normal dargestellt und muß invertiert werden. Entsprechend wird $80 addiert und der folgende Programmteil "$80 subtrahieren" einfach übersprungen (JMP $C010).

Das Programm prüft somit selbstständig, ob die Zeichen dieser Zeile normalisiert oder bereits invertiert sind und kehrt den jeweiligen Zustand um. Wurde die Zeile komplett behandelt, erfolgt mit JMP $C000 ein Neustart. Die Zeile wird abwechselnd invertiert und wieder normalisiert.

Diese dritte Version zeigt, wie schnell Assembler wirklich ist. Die Blinkfrequenz ist dermaßen hoch, daß wir nur ein Flackern wahrnehmen. Die Konsequenz: wir benötigen eine Verzögerungsschleife.

Version 4

```
Bildschirmzeile invertieren/normalisieren
A C000 A2 27        LDX #$27      ;X INITIALISIEREN
A C002 BD 00 04     LDA $0400,X   ;$0400 + X IN AKKU
A C005 30 06        BMI $C00D     ;SPRUNG, WENN NEGATIV
A C007 18           CLC           ;SONST ADD.VORBEREITEN
A C008 69 80        ADC #$80      ;UND $80 ADDIEREN
A C00A 4C 10 C0     JMP $C010     ;SUBTRAKTION ÜBERSPRINGEN
A C00D 38           SEC           ;SUBTRAKTION VORBEREITEN
A C00E E9 80        SBC #$80      ;$80 SUBTRAHIEREN
A C010 9D 00 04     STA $0400,X   ;AKKU NACH $0400 + X
A C013 CA           DEX           ;ZÄHLER VERMINDERN
A C014 10 EC        BPL $C002     ;SPRUNG, WENN POSITIV
Verzögerungsschleife
A C016 A2 FF        LDX #$FF      ;SCHLEIFENZÄHLER INIT.
A C018 20 B3 EE     JSR $EEB3     ;VERZÖGERUNGSROUTINE
A C01B CA           DEX           ;ZÄHLER DEKREMENTIEREN
A C01C D0 FA        BNE $C018     ;SPRUNG, WENN UNGLEICH NULL
A C016 4C 00 C0     JMP $C000     ;NEUSTART DES PROGRAMMS
```

Nachdem eine Zeile komplett invertiert/normalisiert wurde, wird in einer Schleife 255mal die Verzögerungsroutine $EEB3 aufgerufen. C16, Plus/4-Besitzer ersetzen bitte die Adresse $EEb3 durch $E2DC.

Trotz dieser extrem langen Verzögerung ist das Blinken noch ein wenig hektisch. Wenn Sie wollen, können Sie den Verzögerungsschleife in eine weitere Schleife mit Y als Zählregister einpacken.

### 16.3.4. Ein Standardtrick

Die letzte Version des Programms "blinkende Bildschirmzeile" ist für unsere Begriffe bereits recht annehmbar. Assembler-Profis würden jedoch einen bestimmten Programmteil sofort ändern: das Überspringen der Subtraktions-Routine.

Erinnern wir uns: wenn die Zeile noch nicht invertiert ist, addiert der Additionsteil $80 und der folgende Subtraktionsteil wird mit einem JMP-Befehl übersprungen.

So seltsam es auf den ersten Blick erscheinen mag: der Drei-Byte-Befehle JMP kann durch einen kürzeren (zwei Bytes) Branch-Befehl ersetzt werden.

Addiert wird $80 zu einem Wert, der sich irgendwo zwischen $00 und $79 befindet (normales Zeichen). Das Ergebnis liegt daher zwischen $80 und $FF. Da wir genau wissen, daß es niemals (!) den Wert $00 annimmt, können wir JMP durch BNE ersetzen und somit ein Byte einsparen. Die Bedingung "verzweige, wenn ungleich null" ist immer erfüllt!

Dieser Einsatz von Verzweigungen ist ein Standardtrick bei der Assembler-Programmierung. Ein JMP-Befehl kann immer dann durch einen Branch-Befehl ersetzt werden, wenn wir genau wissen, daß ein bestimmtes Flag immer (!) gesetzt oder gelöscht ist. In diesem Fall ist der Branch-Befehl völlig gleichwertig, aus dem bedingten wird ein unbedingter Sprung.

```
                LDA $($0400)                LDA $($0400)
                CMP #$01                    CMP #$01
                BNE $(ADRESSE1)             BNE $(ADRESSE1)
                LDX #$0A                    LDX #$0A
                JMP $(ADRESSE2)             BNE $(ADRESSE2)
(ADRESSE1)      LDX #$0B        (ADRESSE1)  LDX #$0B
(ADRESSE2)      ...             (ADRESSE2)  ...
                ...                          ...
```

Die Aufgabe dieses Programms: der Akkumulator wird mit dem Inhalt der Speicherzelle $0400 geladen. Besitzt $0400 den Wert $01, soll das X-Register mit $0A geladen werden, ansonsten mit $0B.

Im ersten Fall folgt dem Befehl LDX #$0A ein unbedingter Sprung mit JMP (linke Version), um das Laden des X-Registers mit $0B zu übergehen. Der JMP-Befehl kann (rechte Version) problemlos durch den kürzeren Befehl BNE ersetzt werden. Wir

wissen, daß der LDX-Befehl das von BNE getestete Zero-Flag beeinflußt. Da der Wert $0A geladen wird, ist die Bedingung "verzweige, wenn ungleich null" mit Sicherheit (!) erfüllt. In diesem Fall ist BNE ein ebenso unbedingter Sprung wie JMP.

Dieser Standardtrick wird in praktisch jedem Assembler-Programm eingesetzt und im folgenden daher auch von mir verwendet. Nach kurzer Zeit erscheint Ihnen diese Methode wahrscheinlich nicht mehr als Trick, sondern als völlig normale Vorgehensweise.

```
(ADRESSE1) ...              (ADRESSE1) ...
           ...                         ...
           LDX #$FF                    LDX #$FF
(ADRESSE2) ...              (ADRESSE2) ...
           ...                         ...
           DEX                         DEX
           CPX #$0A                    CPX #$0A
           BNE (ADRESSE2)              BNE (ADRESSE2)
           JMP (ADRESSE1)              BEQ (ADRESSE1)
```

In diesem Programm soll ADRESSE1 den Programmanfang kennzeichnen und ADRESSE2 den ersten Befehl in einer Schleife mit X als Zählregister. Der Schleifenzähler X wird, - beginnend mit dem Wert $FF - abwärts gezählt, dekrementiert.

Der Befehl CPX #$0A vergleicht den Inhalt des X-Registers mit $0A. Wenn X nicht $0A enthält, ergibt die bei Vergleichsbefehlen durchgeführte Subtraktion ein Ergebnis ungleich null (Beispiel: X=$FF => $FF - $0A =$F5) und die Bedingung BNE ("verzweige, wenn ungleich null") ist erfüllt, BNE verzweigt zum Schleifenanfang.

Erst wenn X bis $0A heruntergezählt ist, wird die Schleife verlassen. Der Vergleichsbefehl ergibt $00 als Ergebnis ($0A - $0A = $00) und die Sprungbedingung "verzweige, wenn ungleich null" ist nicht erfüllt. Der folgende JMP-Befehl wird ausgeführt und zum Programmanfang gesprungen.

Die rechte Version zeigt, daß JMP durch BEQ ersetzt werden kann. Wir wissen, daß das Ergebnis des Vergleichs gleich null sein muß, wenn dieser Befehl erreicht wird und daß daher BEQ ("verzweige, wenn gleich null") mit Sicherheit verzweigen wird.

Merken wir uns: ein JMP-Befehl kann immer dann durch einen Branch ersetzt werden, wenn wir mit Sicherheit (!) wissen, daß eine bestimmte Bedingung auf alle Fälle erfüllt ist (Ergebnis gleich null, ungleich null, positiv, negativ...). Wir setzen in einem solchen Fall den Branch-Befehl ein, der genau diese Bedingung testet (BEQ, BNE, BPL, BMI...) und sparen ein Byte gegenüber einem JMP.

## 17. Stapeloperationen

In den folgenden Kapiteln werden wir weitere Befehle und Adressierungsarten kennenlernen, mit denen die Assembler-Programmierung erheblich einfacher und vielseitiger wird.

Wie üblich folgt anschließend ein Kapitel mit Demoprogrammen, die den effektiven Einsatz aller bis dahin behandelten Befehle demonstrieren.

In diesem Abschnitt beschäftigen wir uns mit einem bisher vernachlässigten Speicherbereich unseres Rechners, dem Stack. Der Stack oder Stapel ist ein spezieller Speicherbereich, der bei $0100 beginnt und bis $01FF reicht. Der Stapel umfaßt somit die Page (oder Seite) 1 des Speichers, jene Seite, die der Zeropage unmittelbar folgt.

Der Stack ist nach dem sogenannten LIFO-Prinzip organisiert (LIFO=Last in, first out). Im Buch "Programmierung des 6502" von R. Zaks wird der Stack mit einem der häufig im Auto verwendeten Münzspeicher (Zehn-Pfennig-Stücke für Parkuhren) verglichen.

Bei der Datenablage auf dem Stack wird der betreffende Wert von oben auf den Münzspeicher geschoben, alle darunterliegenden Münzen werden nach unten gedrückt. Diese Datenablage

übernimmt der Befehl PHA (push akku, schiebe den Akku (auf den Stack)), der den aktuellen Inhalt des Akkumulators zuoberst auf dem Stack ablegt.

Mit dem Befehl PLA (pull akku, ziehe den Akku (vom Stack)) wird die oberste Münze vom Stack entnommen und in den Akkumulator gebracht.

Da diese Art der Datenablage vom Prozessor verwaltet wird, ist der Stack hervorragend zur kurzfristigen Speicherung von Daten geeignet. Bisher mußten wir den Inhalt des Akkumulators in einer unbenutzten Speicherzelle zwischenspeichern.

```
LDA $0400          ;INHALT VON $0400 LESEN
STA $033C          ;AKKUMULATOR 'RETTEN'
LDA #$FF           ;AKKU MIT $FF LADEN
...
...
LDA $033C          ;URSPRÜNGL.AKKUINHALT ZURÜCKHOLEN
```

Mit den Befehlen PHA und PLA bietet sich die Zwischenspeicherung auf dem Stack an. Mit PHA wird der Akkumulatorinhalt als oberstes Element auf dem Stack abgelegt und - wenn der ursprüngliche Wert wieder benötigt wird - mit PLA zurückgeholt.

```
LDA $0400          ;INHALT VON $0400 LESEN
PHA                ;AKKU AUF STACK 'SCHIEBEN'
LDA #$FF           ;AKKU MIT $FF LADEN
...
...
PLA                ;OBERSTEN WERT VOM STACK 'ZIEHEN'
```

Der große Vorteil des Stacks liegt darin, daß wir uns nicht darum kümmern müssen, welche Speicherzellen unbenutzt sind und zur Zwischenspeicherung von Daten verwendet werden können.

Auf dem Stack können wir bis zu 255 Werte zwischenspeichern und später wieder zurückholen.

```
LDA #$01            ;AKKU MIT $01 LADEN
PHA                 ;AUF STACK SCHIEBEN
LDA #$02            ;AKKU MIT $02 LADEN
PHA                 ;AUF STACK SCHIEBEN
...
...
PLA                 ;$02 VOM STACK ZIEHEN
STA $0400           ;UND NACH $0400
PLA                 ;$01 VOM STACK ZIEHEN
STA $0400           ;UND NACH $0401
```

Beachten Sie bitte bei diesem Beispiel, daß der erste PLA-Befehl das zuletzt auf den Stack geschobene Element holt, der zweite Befehl PLA das vorletzte Element (First in, last out) und erinnern Sie sich an die sehr passende Analogie mit dem Münzspeicher.

Auf den Münzspeicher schieben Sie zuerst eine Ein- und dann eine Zwei-Pfennig-Münze. Die Zwei-Pfennig-Münze befindet sich nun über der Ein-Pfennig-Münze und wird beim Ziehen einer Münze zuerst geholt.

In der Praxis muß sehr oft nicht der Inhalt des Akkumulators, sondern der Inhalt eines der Indexregister gerettet werden. In diesem Fall kopieren wir den Registerinhalt in den Akkumulator und pushen diesen mit PHA.

Wenn wir den Registerwert wieder benötigen, pullen wir den abgelegten Wert mit PLA und übertragen ihn mit TAX oder TAY wieder in das betreffende Register.

```
LDX $0400          ;X MIT INHALT VON $0400 LADEN
TXA                ;X NACH AKKU KOPIEREN
PHA                ;AKKU ALS OBERSTEN WERT AUF STACK SCHIEBEN
...
...
PLA                ;OBERSTEN WERT VOM STACK ZIEHEN
TAX                ;AKKU NACH X KOPIEREN
```

Mit dieser Methode ist es selbstverständlich jederzeit möglich, auch beide Indexregister vorübergehend auf dem Stack zu speichern und, wenn benötigt, die ursprünglichen Werte zurückzuholen.

```
TXA                ;X NACH AKKU
PHA                ;AKKU ALS - VORLÄUFIG - OBERSTES ELEMENT AUF
STACK
TYA                ;Y NACH AKKU
PHA                ;AKKU ALS OBERSTES ELEMENT AUF STACK
...
...
PLA                ;OBERSTES STACK-ELEMENT HOLEN
TAY                ;UND NACH Y
PLA                ;NEUES OBERSTES ELEMENT HOLEN
TAX                ;UND NACH X
```

Beachten Sie die Reihenfolge: Der Inhalt des X-Registers wird als oberstes Element auf den Stack gebracht, anschließend jedoch durch den folgenden PHA-Befehl (Y-Registerinhalt) nach unten gedrückt.

Beim Holen beider Werte wird daher zuerst der ursprüngliche Inhalt des Y-Registers geholt, die darunterliegenden Werte rücken nach und anschließend wird mit dem zweiten PLA-Befehl der ursprüngliche X-Inhalt geholt, der sich nun an der Stapelspitze befindet.

Übrigens: die beiden Befehle TXA und PHA sind nicht länger als der konventionelle Befehl STX $033C, sogar kürzer. Beides sind Ein-Byte-Befehle, im Gegensatz zum Drei-Byte-Befehl

STX $033C. Gleiches gilt für PLA/TAX im Vergleich zu LDX $033C.

Ab und zu werden wir in den folgenden Kapiteln zwei weitere Befehle benötigen, die mit dem Stack arbeiten, PHP und PLP.

PHP (push processor) schiebt das Status-Register des Prozessors als oberstes Element auf den Stack. PLP (pull processor) zieht das oberste Element vom Stack und überträgt es in das Status-Register.

Mit der Kombination dieser Befehle können wir jederzeit den aktuellen Flagzustand auf den Stack retten und bei Gelegenheit zurückholen:

```
          LDX #$FF
(ADRESSE) ...
          ...
          DEX
          PHP
          LDA #$0A
          PLP
          BNE (ADRESSE)
```

In diesem - nicht unbedingt sehr sinnvollen - Beispiel wird nach der Dekrementierung des Zählregisters X der aktuelle Flagzustand - das Status-Register - auf den Stack gerettet, da die Flags durch den Befehl LDA #$0A verändert werden.

Vor dem Testen des Zero-Flags wird der ursprüngliche Inhalt des Status-Registers mit PLP wieder vom Stack geholt, bevor der Test des Zero-Flags mit BNE erfolgt.

Beachten Sie bei Datenspeicherung auf dem Stack immer das LIFO-Prinzip (das zuletzt auf den Stack gebrachte Element wird als erstes geholt)!

```
LDA #$01
PHA                 ;$01 'PUSHEN'
LDA #$02
```

```
PHA                 ;$02 'PUSHEN'
LDA #$03
PHA                 ;$03 'PUSHEN'
...
...
PLA                 ;$03 'PULLEN'
STA $0400
PLA                 ;$02 'PULLEN'
STA $0401
PLA                 ;$01 'PULLEN'
STA $0400
```

Wie wir noch sehen werden, ist es zudem außerordentlich wichtig, daß zu jedem PHA auch ein PLA gehört, der das Element zu einem späteren Zeitpunkt wieder vom Stack holt!

Wenn Sie noch der Methode vorgehen: "Alles, was später eventuell gebraucht werden könnte, rette ich auf den Stack. Vielleicht hole ich diese Werte wieder irgendwann vom Stack, vielleicht auch nicht", hängt sich Ihr Programm früher oder später auf (da der Stack überfüllt ist), d.h., Sie müssen den Rechner ausschalten oder aber - wenn Sie sich inzwischen einen solchen angeschafft haben - den Reset-Schalter betätigen.

## 18. Die indirekte Adressierung

Wir kennen die implizite (TAX, PHA..), die unmittelbare (LDA #$FF..), die absolute (LDX $0400), die Zeropage- (LDA $F0) und die indizierte (LDA $0400,X) Adressierung.

Zwei Adressierungsarten fehlen uns noch, die indirekte und die indirekt indizierte Adressierung. Die Erläuterung dieser Adressierungsarten wurde bis zu diesem Zeitpunkt zurückgestellt, da beide sehr ungewöhnlich und eher zu verstehen sind, wenn bereits Assembler-Grundkenntnisse (die Sie inzwischen besitzen) vorhanden sind.

Indirekte Adressierung bedeutet, daß die zu verwendende Adresse nicht direkt (wie bei LDA $0400), sondern über einen Umweg angegeben wird.

Dem jeweiligen Befehl wird eine Adresse angegeben, die die eigentlich gewünschte Adresse enthält. Eine Analogie: wir benötigen eine Telefonnummer und rufen die Telefonauskunft an. Im Telefonbuch finden wir die Nummer der Auskunft. Die Nummer der Auskunft ist ein Umweg, über den wir letztendlich die gewünschte Nummer erfahren.

Entsprechend verläuft die indirekte Adressierung. Dem jeweiligen Befehl geben wir die Adresse einer Speicherzelle an, die die eigentlich gewünschte Adresse enthält.

```
Zieladresse in $033C/$033D ablegen
      A C000 A9 0A        LDA #$0A      ;LOW-BYTE VON $C00A
      A C002 8D 3C 03     STA $033C     ;NACH $033C SCHREIBEN
      A C005 A9 C0        LDA #$C0      ;HIGH-BYTE VON $C00A
      A C007 8D 3D 03     STA $033D     ;NACH $033D SCHREIBEN
      Hauptprogramm
      A C00A EE 20 D0     INC $D020     ;$D020 INKREMENTIEREN
      A C00D 6C 3C 03     JMP ($033C)   ;JMP INDIREKT !!!
```

Dieses Programm inkrementiert in einer Endlosschleife die Farbe des Bildschirmrahmens ($D020 = Rahmenfarbe beim C64).

Die ersten vier Befehle legen die Adresse $C00A in Form von Low-Byte/High-Byte in den Speicherzellen $033C (Inhalt $0A) und $033D (Inhalt $C0) ab. Anschließend wird die Rahmenfarbe inkrementiert.

Der folgende JMP-Befehl ist ein indirekter Sprung. In Klammern (!) wird die Adresse angegeben, an der die eigentliche Sprungadresse zu finden ist!

Das heißt: der JMP-Befehl springt nicht (!) zur Adresse $033C, sondern zu jener Adresse, die in $033C und $033D in der Form Low-Byte/High-Byte abgelegt ist.

Mit dem vorbereitenden Teil wurde die Adresse $C00A abgelegt, die Adresse des Befehls INC $D020. Der JMP-Befehl findet ab der Adresse $033C daher die eigentliche Sprungadresse $C00A vor, zu der gesprungen wird.

In diesem Beispiel ist JMP($033C) somit von der Funktion her identisch mit dem direkten Sprungbefehl JMP $C00A.

Merken wir uns: dem indirekten Sprung JMP (...) wird ein sogenannter Vektor oder Zeiger angegeben, der die eigentliche Zieladresse enthält. Ein Vektor besteht aus zwei aufeinanderfolgenden Speicherzellen, die eine 16-Bit-Adresse - die eigentliche Sprungadresse - in der Form Low-Byte/High-Byte enthalten, dem Adreßformat.

Die indirekte Adressierung ist nur mit dem JMP-Befehl möglich und wird uns nicht weiter interessieren. Diese Adressierungsart wurde erläutert, da sie die Grundlage für die eminent wichtige indirekt-indizierte Adressierung bildet, die im folgenden Abschnitt beschrieben wird.

## 19. Die indirekt-indizierte Adressierung

Die indirekt-indizierte Adressierung ist eine Kombination aus der soeben beschriebenen indirekten und der bereits bekannten indizierten Adressierung. Sie existiert in zwei Varianten, der Vorindizierung und der Nachindizierung. Wir werden uns fast ausschließlich mit der Nachindizierung beschäftigen, die wesentlich wichtiger ist als die Vorindizierung. Ein Beispiel:

```
LDA ($FB),Y
```

Die Klammer um die Zeropage-Adresse $FB zeigt an, daß es sich um einen Zeiger (Vektor) handelt, daß $FB und die folgende Speicherzelle $FC eine Adresse enthalten. Nehmen wir an, in $FB befindet sich der Wert $00 und in $FC der Wert $C0.

```
Speicherzellen   Inhalt

$FB              $00
$FC              $C0
```

$FB/$FC enthalten somit die 16-Bit-Adresse $C000 (Low-Byte $00/High-Byte $C0). Ähnlich wie bei der indizierten Adressierung wird zu dieser Adresse der Inhalt des Y-Registers addiert. Angenommen, Y enthält den Inhalt $04. Dann ergibt sich als endgültige Adresse $C004 ($C000 + $04 = $C004).

Der Befehl LDA ($FB),Y lädt den Inhalt von Speicherzelle $C004 in den Akkumulator, wenn $FB/$FC die Adresse $C000 und das Y-Register den Wert $04 enthalten.

Diese Variante der indirekt-indizierten Adressierung heißt nachindiziert, weil zuerst die indirekt angegebene Zieladresse gelesen und danach (!) der Inhalt des Y-Registers addiert wird.

Wichtig: die in Klammern angegebene Adresse - der Zeiger - muß sich in der Zeropage befinden! Außerdem kann im Gegensatz zur indizierten Adressierung nur das Y-Register zur Indizierung eingesetzt werden, LDA ($FB),X ist nicht zulässig!

Mit dieser Adressierungsform können wir höchst elegant auf beliebige Speicherbereiche zugreifen, auf Blöcke. Da sich der Zeiger jedoch in der Zeropage befinden muß, sind wir bei dieser Adressierungsart auf wenige Speicherzellen beschränkt. Folgende Zeropageadressen sind unbenutzt und eignen sich zur Ablage von Zeigern:

```
C64:          $FB-$FE (4 Byte)
C16, Plus/4:  $D8-$E8 (17 Byte)
C128:         $FA-$FE (5 Byte)
```

In der Praxis benötigt man niemals mehr als zwei Zeiger gleichzeitig. Einigen wir uns auf folgende Konvention: für einen Zeiger verwenden wir $FB/$FC, benötigen wir einen zweiten Zeiger, benutzen wir zusätzlich $FD/$FE.

C16, Plus/4-Besitzer ersetzen bitte in allen folgenden Programmen, die Zeiger verwenden, $FB/$FC durch $D8/$D9 und $FD/$FE durch $DA/$DB.

Nun jedoch zum praktischen Einsatz dieser Adressierungsart. Erinnern Sie sich: mit der indizierten Adressierung konnten wir problemlos auf bis zu 256 aufeinanderfolgende Speicherzellen zugreifen, indem wir den Wert des Indexregisters mit einer Schleife veränderten.

```
          LDA #$01
          LDX #$00
(ADRESSE) STA $0400,X
          DEX
          BNE (ADRESSE)
```

Dieses Programm schreibt in die ersten 256 (Schleife inclusive (!) null) Speicherzellen des Video-RAM den Bildschirmcode des Zeichens A. Um eine Schleife 256mal (inclusive der null) zu durchlaufen, benötigten wir bislang den Befehl BCS und einen zusätzlichen Vergleichsbefehl:

```
          LDA #$01
          LDX #$FF
(ADRESSE) STA $0400,X
          DEX
          CPX #$FF
          BCS (ADRESSE)
```

Mit einem kleinen Trick, der in der ersten Version des Programms angewendet wird, verringert sich dieser Aufwand. Das Zählregister X wird mit $00 initialisiert, nicht wie üblich mit $FF. Zuerst wird daher auf $0400 zugegriffen. Anschließend wird X dekrementiert und es erfolgt der bei Dekrementierungen bekannte Unterlauf von $00 zu $FF.

Im zweiten Durchlauf wird daher auf das entgegengesetzte Ende des Blocks zugegriffen, auf $04FF und in den folgenden Durchgängen wie üblich nacheinander auf $04FE, $04FD, $04FC,...,$0401.

Zuletzt wird $0401 behandelt, da die folgende Dekrementierung zum Ergebnis $00 führt ($01 - $00 = $00) und die Bedingung BNE ("verzweige, wenn ungleich null") nicht erfüllt ist.

Daß $0400 nicht mehr behandelt wird, ist jedoch nicht tragisch. Auf diese Speicherzelle wurde bereits beim ersten Durchlauf zugegriffen.

Mit diesem kleinen Trick können wir problemlos einen kompletten Block von 256 Zeichen behandeln, inclusive des problematischen nullten Bytes.

Um alle vier Bildschirmseiten mit dem Zeichen A zu füllen, benötigten wir bisher ein Programm wie das folgende:

```
            LDA #$01
            LDX #$00
(ADRESSE)   STA $0400,X
        STA $0500,X
            STA $0600,X
            STA $0700,X
            DEX
            BNE (ADRESSE)
```

Diese Schleife behandelt zuerst die Speicherzellen $0400, $0500, $0600 und $0700, also die ersten Speicherzellen jeder Seite. Anschließend wird das X-Register dekrementiert und erhält den Wert $FF ($00 - $01 = $FF).

Im folgenden Durchgang werden daher $04FF, $05FF, $06FF und $07FF behandelt, die jeweils letzten Speicherzellen der vier Bereiche. Nach erfolgter Dekrementierung des X-Registers enthält es den Wert $FE und die jeweils vorletzten Speicherzellen werden mit dem Code für A gefüllt ($04FE, $05FE, $06FE, $07FE).

Im letzten Durchgang hat X den Wert $01 und die Speicherzellen $0401, $0501, $0601 und $0701 werden behandelt.

Alle vier Seiten des Bildschirmspeichers ($0400-$07FF) sind mit dem Bildschirmcode des Zeichens A gefüllt worden.

Mit der indirekt-indizierten Adressierung wird das Programm nicht unbedingt kürzer, jedoch weitaus eleganter:

Bildschirm mit A's füllen

```
Zeiger auf Anfang des Bildschirmspeichers nach $FB/$FC
A C000 A0 00        LDA #$00          ;LOW-BYTE VON $0400
A C002 85 FB        STA $FB           ;NACH $FB
A C004 A0 04        LDA #$04          ;HIGH-BYTE VON $0400
A C006 85 FC        STA $FC           ;NACH $FC
Schleifenzähler und Akkumulator initialisieren
A C008 A9 01        LDA #$01          ;$01 = CODE FÜR 'A'
A C00A A2 04        LDX #$04          ;X=ZÄHLER ÄUSSERE SCHLEIFE
A C00C A0 00        LDY #$00          ;Y=ZÄHLER INNERE SCHLEIFE
Hauptprogramm
A C00E 91 FB        STA ($FB),Y       ;'A' NACH ($FB),Y
A C010 88           DEY               ;ZÄHLER DEKREMENTIEREN
A C011 D0 FB        BNE $C00E         ;SPRUNG, WENN UNGLEICH NULL
A C013 E6 FC        INC $FC           ;SONST ZEIGER AUF NÄCHSTE PAGE
A C015 CA           DEX               ;ZÄHLER DEKREMENTIEREN
A C016 D0 F4        BNE $C00C         ;SPRUNG, WENN UNGLEICH NULL
A C018 00           BRK               ;PROGRAMMENDE
```

Im vorbereitenden Teil wird die Startadresse des Bildschirms (C64: $0400) in $FB/$FC in der Form Low-Byte/High-Byte abgelegt, die Zeiger auf dem Bildschirm erzeugt. Zur Vorbereitung gehört ebenfalls das Laden des Akkumulators mit $01, dem Bildschirmcode des Zeichens A, und die Initialisierung der Schleifenzähler X und Y mit $04 bzw. $00.

Der Hauptteil besteht aus zwei ineinandergeschachtelten Schleifen. Die innere Schleife mit Y als Zählregister ist für die Behandlung eines Blocks mit 256 Zeichen zuständig.

Betrachten wir den ersten Durchgang: Y enthält den Wert $00. Der Inhalt des Akkumulators ($01=A) wird in die Speicherzelle

$0400 geschrieben (der Zeiger $FB/$FC weist auf $0400; zu $0400 wird Y=$00 addiert).

Y wird dekrementiert (DEY) und hat nun den Wert $FF, d.h., BNE verzweigt zu Adresse $C00E. Der Befehl STA ($FB),Y bezieht sich nun auf die Speicherzelle $04FF (Zeiger in $FB/$FC weist auf $0400; Y gleich $FF => $04FF), die letzte Speicherzelle der ersten Bildschirmseite.

Y wird erneut dekrementiert (=> Y=$FE) und zum Beginn der inneren Schleife verzweigt. Der Bildschirmcode des Zeichens A wird in die Speicherzelle $04FE geschrieben und so weiter.

Im letzten Durchgang der inneren Schleife besitzt Y den Wert $01, der Befehl STA ($FB),Y bezieht sich daher auf $0401, die vorletzte Zelle von Bildschirmseite 1 ($0400 wurde bereits - im ersten Durchgang - behandelt).

DEY führt nun zum Ergebnis $00 und BNE verzweigt nicht mehr, die innere Schleife wird verlassen. Entscheidend an diesem Programm ist der Befehl INC $FC. $FC ist das High-Byte des Zeigers, der momentan auf die Adresse $0400 weist. Wird das High-Byte dieses Zeigers inkrementiert, weist er auf den Beginn der zweiten Bildschirmseite, also auf den nächsten zu behandelnden Block. Aus dem Zeiger $00/$04 (=$0400 in der Form Low/High), in den Speicherzellen $FB/$FC, wird durch die Inkrementierung von $FC der neue Zeiger $00/$05 (=$0500 in der Form Low/High).

Das X-Register wird dekrementiert und - da X ungleich null - mit BNE zu $C00C verzweigt. Nachdem das Y-Register wieder mit $00 intialisiert wurde, wird die innere Schleife erneut 256mal durchlaufen. Diesmal wird jedoch statt auf den Bereich $0400-$04FF auf $0500-$05FF zugegriffen.

Die äußere Schleife wird insgesamt viermal durchlaufen (LDX #$04), um alle vier Seiten des Bildschirms zu behandeln.

Sie sehen, mit der indirekt-indizierten Adressierung kann sehr elegant auf beliebig große Blöcke zugegriffen werden. Nachdem

ein Block mit 256 Byte behandelt wurde, wird das High-Byte des Zeigers inkrementiert und in der inneren Schleife der folgende 256-Byte-Block bearbeitet.

Noch eine Bemerkung: am Ende der äußeren Schleife wird zu $C00C verzweigt, um durch erneute Initialisierung des Zählregisters Y mit dem Startwert $00 die nächsten 256 Durchgänge der inneren Schleife vorzubereiten.

Wenn Sie sich das Programm näher anschauen, stellen Sie jedoch fest, daß Y nach dem Verlassen der inneren Schleife sowieso den Wert $00 besitzt und eine Initialisierung mit diesem Wert überflüssig ist. Anstatt zu $C00C kann daher gleich zu $C00E (dem eigentlichen Beginn der inneren Schleife) verzweigt werden.

Ich muß zugeben, daß das Demoprogramm zur Behandlung des Bildschirms mit der indizierter Adressierung kürzer ist, da das Einrichten des Zeigers in $FB/$FC entfällt.

Bedenken Sie jedoch, wie groß der Aufwand mit indizierter Adressierung wird, wenn umfangreiche Speicherbereiche bearbeitet werden, z.B. der Bereich $8000-$8FFF (vier Kilobyte).

*Indizierte Adressierung*

```
        LDX #$00
(ADRESSE) STA $8000,X
        STA $8100,X
        STA $8200,X
        STA $8300,X
        STA $8400,X
        STA $8500,X
        STA $8600,X
        STA $8700,X
        STA $8800,X
        STA $8900,X
        STA $8A00,X
        STA $8B00,X
        STA $8C00,X
```

*Indirekt-indizierte Adressierung*

```
        LDA #$00
        STA $FB
        LDA #$80
        STA $FC
        LDX #$10
        LDY #$00
(ADRESSE) STA ($FB),Y
        DEY
        BNE (ADRESSE)
        INC $FC
        DEX
        BNE (ADRESSE)
```

```
STA $8D00,X
STA $8E00,X
STA $8F00,X
DEX
BNE (ADRESSE)
```

Bei der indirekt-indizierten Adressierung ist es gleichgültig, ob nur ein Block mit 256 Bytes bearbeitet wird, oder aber beliebig viele Blöcke. Das X-Reister wird mit der jeweiligen Blockanzahl geladen, der Rest des Programms bleibt unverändert.

Der Umgang mit der indirekt-indizierten Adressierung:

1. In der Zeropage wird ein Zeiger auf den Beginn des Speicherbereichs in zwei aufeinanderfolgenden Speicherzellen abgelegt (Low-Byte, dann High-Byte).

2. Der Zähler der äusseren Schleife wird mit der Anzahl der zu bearbeitenden 256-Byte-Blöcke geladen.

3. Der Zähler der inneren Schleife wird initialisiert.

4. Die innere Schleife wird durchlaufen und greift mit indirekt-indizierter Adressierung auf die erste Seite des betreffenden Speicherbereichs zu.

5. Wenn die innere Schleife verlassen wird (ein Block ist komplett bearbeitet), wird das High-Byte des Zeigers inkrementiert. Der Zeiger weist nun genau 256 Byte weiter, also auf den Beginn des nächsten Blocks.

6. Der Blockzähler X wird dekrementiert und - wenn ungleich null - zum Beginn der inneren Schleife verzweigt. Die nächsten 256 Byte werden bearbeitet.

## 19.1. Übungsprogramme zur indirekt-indizierten Adressierung

Sie kennen nun die flexibelste Form der Adressierung, die unser Rechner besitzt. Diese Adressierungsart ist dermaßen vielseitig verwendbar, daß wohl kaum ein größeres Assembler-Programm ohne die indirekt-indizierte Adressierung auskommt.

Außer LDA können folgende Befehle mit der beschriebenen Adressierung verwendet werden: ADC, CMP, SBC und STA.

Da diese Adressierungsart so eminent wichtig ist, stelle ich Ihnen in diesem Abschnitt einige Übungsprogramme vor, die außer STA auch LDA und CMP mit indirekt-indizierter Adressierung verwenden.

### 19.1.1. Selektierte Zeichen invertieren

Angenommen, Sie wollen ein bestimmtes Zeichen überall, wo es auf dem Bildschirm vorhanden ist, invertieren. Ohne die indirekt-indizierte Adressierung wird Ihnen dieses Problem einige Kopfschmerzen bereiten, mit dieser Adressierungsart ist es dagegen sehr einfach zu lösen:

```
Zeiger einrichten
A C000 A0 00        LDA #$00      ;LOW-BYTE VON $0400
A C002 85 FB        STA $FB       ;NACH $FB
A C004 A0 04        LDA #$04      ;HIGH-BYTE VON $0400
A C006 85 FC        STA $FC       ;NACH $FC
Zähler initialisieren
A C008 A2 04        LDX #$04      ;ZÄHLER ÄUSSERE SCHLEIFE INIT.
A C00A A0 00        LDY #$00      ;ZÄHLER INNERE SCHLEIFE INIT.
Schleifenanfang
A C00C B1 FB        LDA ($FB),Y   ;AKKU MIT ($FB),Y LADEN
A C00E C9 20        CMP #$20      ;MIT $20(=SPACE) VERGLEICHEN
A C010 D0 04        BNE $C016     ;SPRUNG, WENN UNGLEICH
A C012 A9 A0        LDA #$A0      ;AKKU MIT $A0(=SPACE INVERS) LADEN
A C014 91 FB        STA ($FB),Y   ;UND NACH ($FB),Y SCHREIBEN
A C016 88           DEY           ;ZÄHLER INNEN DEKREMENTIEREN
A C017 D0 F3        BNE $C00C     ;SPRUNG, WENN UNGLEICH NULL
```

```
A C019 E6 FC      INC $FC       ;SONST ZEIGER AUF NÄCHSTE SEITE
A C01B CA         DEX           ;ZÄHLER AUSSEN DEKREMENTIEREN
A C01C D0 EE      BNE $C00C     ;SPRUNG, WENN UNGLEICH NULL
A C01E 00         BRK           ;PROGRAMMENDE
```

Der Aufbau unterscheidet sich kaum vom vorhergehenden Programm. Der einzige prinzipielle Unterschied besteht darin, daß der Branch-Befehl am Ende der äußeren Schleife nicht zur - überflüssigen - Neuinitialisierung des Y-Registers, sondern sofort zum Beginn der inneren Schleife führt.

Wirkliche Unterschiede ergeben sich in der inneren Schleife. Diese Schleife besitzt die Aufgabe, jede Speicherzelle des Bildschirms auf den Inhalt $20 (Leerzeichen oder Space) zu überprüfen.

Mit LDA ($FB),Y wird indirekt-indiziert auf die Speicherzellen zugegriffen. Wenn der Inhalt der untersuchten Zelle gleich $20 ist, wird der folgende Branch-Befehl (BNE $C016) nicht ausgeführt. Der Akkumulator wird mit $A0 geladen ($A0 = $20 + $80 = inverses Leerzeichen) und dieser Bildschirmcode ebenfalls indirekt-indiziert in die betreffende Speicherzelle zurückgeschrieben.

Auf diese Weise werden alle vier Bildschirmseiten behandelt (LDX #$04) und alle Leerzeichen invertiert.

Welche Zeichen invertiert werden, bestimmen Sie. Wollen Sie z.B. alle A's invertieren, ersetzen Sie CMP #$20 durch CMP #$01 und LDA #$80 durch LDA #$81 ($01=Bildschirmcode für inverses A).

Durch Variieren dieser beiden Befehle können Sie ein beliebiges Zeichen auf dem gesamten Bildschirm durch ein ebenfalls beliebiges anderes Zeichen ersetzen (mit CMP #$20 und LDA #$01 werden alle Leerzeichen durch A's ersetzt).

### 19.1.2. Windowing

Jedes moderne Programm arbeitet heutzutage mit Windowing, d.h., mit Fenstern. Darunter ist zu verstehen, daß in einem beliebigen Bildschirmbereich z.B. ein Hilfstext oder ein Menü eingeblendet und nach getaner Arbeit wieder ausgeblendet wird. Nach dem Ausblenden ist der ursprüngliche Bildschirmzustand unverändert widerhergestellt.

Dieses Windowing kann selbstverständlich auch mit einem BASIC-Programm durchgeführt werden. Über den aktuellen Bildschirminhalt wird mit PRINT-Befehlen ein Hilfstext gelegt. Wird der Hilfstext nicht mehr benötigt, wird der Bildschirm gelöscht und mit einer weiteren Anzahl von PRINT-Befehlen der urspüngliche Inhalt (Text, Adresse etc.) wieder auf den Bildschirm geschrieben.

Mit blitzschnellem Ein- bzw. Ausblenden hat diese Vorgehensweise allerdings nichts gemeinsam. Der ursprüngliche Bildschirminhalt wird mit PRINT-Befehlen sehr gemütlich Zeile für Zeile wiederhergestellt.

Mit der indirekt-indizierten Adressierung sind wir in der Lage, echtes Windowing in Assembler zu verwirklichen. Wir retten den kompletten Bildschirminhalt (vier Seiten a 256 Zeichen) in einen freien Speicherbereich und kopieren ihn auf Kommando wieder zurück.

Für das folgende Programm benötigen wir zwei Zeiger. Der erste Zeiger weist auf die Startadresse des Bildschirmspeichers ($0400 beim C64, $0C00 beim C16, Plus/4), der zweite Zeiger auf die erste Speicherzelle des freien Bereichs.

Beim C64 verwenden wir den Bereich $C000-$C3FF, der sich hinter dem eingentlichen Programm befindet.

Wir verwenden ein Unterprogramm, das mit RTS endet und von BASIC aus mit dem SYS-Befehl aufgerufen wird. Das Unterprogramm besitzt zwei verschiedene Einsprungadressen, die über

die jeweilige Funktion entscheiden (Bildschirm retten/geretteten Inhalt zurückkopieren).

```
Version 1

Zeiger einrichten (Einsprung 1)
A C000 A9 00      LDA #$00        ;LOW-BYTE VON $0400
A C002 85 FB      STA $FB         ;NACH $FB
A C004 A9 04      LDA #$04        ;HIGH-BYTE VON $0400
A C006 85 FC      STA $FC         ;NACH $FC
A C008 A9 00      LDA #$00        ;LOW-BYTE VON $C400
A C00A 85 FD      STA $FD         ;NACH $FD
A C00C A9 C4      LDA #$C4        ;HIGH-BYTE VON $C400
A C00E 85 FE      STA $FE         ;NACH $FE
A C010 4C 23 C0   JMP $C023       ;FOLGENDEN TEIL ÜBERSPRINGEN
Zeiger einrichten (Einsprung 2)
A C013 A9 00      LDA #$00        ;LOW-BYTE VON $C400
A C015 85 FB      STA $FB         ;NACH $FB
A C017 A9 C4      LDA #$C4        ;HIGH-BYTE VON $C400
A C019 85 FC      STA $FC         ;NACH $FC
A C01B A9 00      LDA #$00        ;LOW-BYTE VON $0400
A C01D 85 FD      STA $FD         ;NACH $FD
A C01F A9 04      LDA #$04        ;HIGH-BYTE VON $0400
A C021 85 FE      STA $FE         ;NACH $FE
Bereich kopieren
A C023 A2 04      LDX #$04        ;BLOCKZÄHLER: 4 BLÖCKE KOPIEREN
A C025 A0 00      LDY #$00        ;Y INITIALISIEREN
A C027 B1 FB      LDA ($FB),Y     ;ZEICHEN LESEN
A C029 91 FD      STA ($FD),Y     ;ZEICHEN KOPIEREN
A C02B 88         DEY             ;ZÄHLER INNEN DEKREMENTIEREN
A C02C D0 F9      BNE $C027       ;KOMPLETTEN BLOCK KOPIERT?
A C02E E6 FC      INC $FC         ;HIGH-BYTE VON ZEIGER 1 INKREM.
A C030 E6 FE      INC $FE         ;HIGH-BYTE VON ZEIGER 2 INKREM.
A C032 CA         DEX             ;BLOCKZÄHLER DEKREMENTIEREN
A C033 D0 F2      BNE $C027       ;ALLE BLÖCKE KOPIERT?
A C035 60         RTS             ;ZURÜCK NACH BASIC!
```

Der umfangreichste Teil dieses Programms besteht aus dem Einrichten der beiden Zeiger. Der Einsprung 1 (SYS 49152) richtet in $FB/$FC einen Zeiger auf die Startadresse des Bildschirm-

speichers und in $FD/$FE einen zweiten Zeiger auf den freien Speicherbereich $C400 ein. Wenn die beiden Zeiger eingerichtet sind, wird der folgende Programmteil übersprungen (JMP $C023).

An der zweiten Einsprungadresse $C013 (SYS 49171) werden ebenfalls zwei Zeiger eingerichtet, jedoch mit vertauschten Adressen ($C400 in $FB/$FC und $0400 in $FD/$FE).

Zu Beginn des Hauptprogramms erfolgt die bereits bekannte Initialisierung der Zählregister. Der Blockzähler X wird mit dem Wert $04 geladen, da auf insgesamt vier Blöcke mit je 256 Byte zugegriffen werden soll. Das Y-Register wird mit $00 initialisiert und die innere Schleife beginnt.

Betrachten wir zuerst den Fall, daß der Benutzer das Programm mit SYS 49152 aufrief. Der Einsprung an Adresse $C000 führte zur Einrichtung folgender Zeiger:

```
$FB/$FC = Zeiger auf $0400
$FD/$FE = Zeiger auf $C400
```

Im ersten Durchgang lädt der Befehl LDA ($FB),Y den Akkumulator mit dem Inhalt der Speicherzelle $0400 (Zeigerinhalt + Y-Register => $0400 + $00). Der Inhalt dieser Speicherzelle wird mit STA ($FD),Y nach $C400 kopiert (Zeigerinhalt + Y-Register => $C400 + $00).

Das Y-Register wird dekrementiert (=> Y=$FF) und - da das Ergebnis $FF die Bedingung BNE $C027 erfüllt - die Schleife erneut durchlaufen.

Aufgrund des Y-Wertes $FF wird nun jedoch auf das entgegengesetzte Blockende zugegriffen. Der Akkumulator wird mit dem Inhalt der Speicherzelle $04FF ($0400 + $FF) geladen und nach $C4FF ($C400 + $FF) kopiert.

In den folgenden Durchgängen wird Y dekrementiert und der Reihe nach $04FE nach $C4FE, dann $04FD nach $C4FD und

zuletzt $0401 nach $C401 kopiert. Der erste 256-Byte-Block ($0400-$04FF) wurde vollständig nach $C400-$C4FF kopiert.

Die High-Bytes beider Zeiger werden nun inkrementiert und weisen in den folgenden Durchgängen auf die Startadresse des nächsten Blocks ($0500 und $C500). Die nächsten 256 Durchgänge der inneren Schleife kopieren daher den Inhalt des Blocks $0500-$05FF nach $C500-$C5FF.

Der Vorgang wiederholt sich insgesamt viermal. Dann besitzt das X-Register - das nach jeder Blockübertragung dekrementiert wird - den Inhalt $00 und die Bedingung BNE $C027 ist nicht mehr erfüllt.

Der gesamte Bildschirmspeicher wurde kopiert und mit RTS erfolgt die Rückkehr nach BASIC.

Analog verläuft die Wiederherstellung des ursprünglichen Bildschirminhalts. Das Programm wird von BASIC aus mit SYS 49171 (=$C013) aufgerufen und startet vom zweiten Einsprungpunkt aus. Die Zeiger werden nun genau umgekehrt eingerichtet:

```
$FB/$FC = Zeiger auf $C400
$FD/$FE = Zeiger auf $0400
```

Der Kopiervorgang verläuft daher von $C400-$C7FF nach $0400-$07FF. Der zuvor gerettete Bildschirminhalt wird in den Bildschirmspeicher zurückkopiert.

Bevor ich Ihnen ein kleines BASIC-Programm vorstelle, das die praktische Anwendung dieses Assembler-Programms demonstriert, zeige ich Ihnen, wie dieses Programm weitaus kürzer und eleganter geschrieben wird.

1. Die Befehlsfolge:

```
LDA #$00
STA $FB
LDA #$04
STA $FC
LDA #$00
STA $FD
...
...
```

kann erheblich verkürzt werden. Sowohl nach $FB als auch nach $FC soll der Wert $00 geschrieben werden. Es genügt vollkommen, den Akkumulator einmal (!) mit $00 zu laden und dann mit zwei aufeinanderfolgenden Befehlen in beide Speicherzellen zu übertragen. Gleiches gilt für die Zeigereinrichtung am zweiten Einsprungpunkt.

2. Unabhängig vom Einsprungpunkt (1 oder 2) wird auf alle Fälle in $FB und in $FD das Low-Byte $00 übertragen. Es bietet sich daher an, nur die Speicherung der High-Bytes vom Einsprungpunkt abhängig zu machen und erst nach der Zusammenführung der Einsprungpunkte die identischen Low-Bytes in $FB und $FD zu speichern.

3. Beim Einsprung an Adresse $C000 wird nach dem Einrichten der Zeiger der folgende Programmteil mit JMP $C023 übersprungen. Der Drei-Byte-Befehl JMP kann durch den Zwei-Byte-Befehl BNE ersetzt werden. Der letzte Befehl, der das Zero-Flag beeinflußte, das BNE testet, war LDA #$90 (Ergebnis ungleich null). Wir wissen daher mit absoluter Sicherheit, daß die Bedingung BNE ("verzweige, wenn ungleich null") erfüllt ist und der folgende Teil auch mit diesem Branch-Befehl übersprungen wird.

Version 2

```
Zeiger-High-Bytes einrichten (Einsprung 1)
A C000 A9 04        LDA #$04          ;HIGH-BYTE VON $0400
A C002 85 FC        STA $FC           ;NACH $FC
A C004 A9 C4        LDA #$C4          ;HIGH-BYTE VON $C400
A C006 85 FE        STA $FE           ;NACH $FE
A C008 D0 08        BNE $C012         ;FOLGENDEN TEIL ÜBERSPRINGEN
Zeiger-High-Bytes einrichten (Einsprung 2)
A C00A A9 C4        LDA #$C4          ;HIGH-BYTE VON $C400
A C00C 85 FC        STA $FC           ;NACH $FC
A C00E A9 04        LDA #$04          ;HIGH-BYTE VON $0400
A C010 85 FE        STA $FE           ;NACH $FE
Gemeinsame Low-Bytes einrichten
A C012 A9 00        LDA #$00          ;LOW-BYTE $00
A C014 85 FB        STA $FB           ;NACH $FB
A C016 85 FD        STA $FD           ;UND NACH $FD
Bereich kopieren
A C018 A2 04        LDX #$04          ;4 BLÖCKE KOPIEREN
A C01A A0 00        LDY #$00          ;ZÄHLER INNEN INITIALISIEREN
A C01C B1 FB        LDA ($FB),Y       ;ZEICHEN LESEN
A C01E 91 FD        STA ($FD),Y       ;ZEICHEN KOPIEREN
A C020 88           DEY               ;ZÄHLER INNEN DEKREMENTIEREN
A C021 D0 F9        BNE $C01C         ;KOMPLETTEN BLOCK KOPIERT?
A C023 E6 FC        INC $FC           ;HIGH-BYTE ZEIGER 1 INKREM.
A C025 E6 FE        INC $FE           ;HIGH-BYTE ZEIGER 2 INKREM.
A C027 CA           DEX               ;BLOCKZÄHLER DEKREMENTIEREN
A C028 D0 F2        BNE $C01C         ;ALLE BLÖCKE KOPIERT?
A C02A 60           RTS               ;ZURÜCK NACH BASIC
```

Dieses Programm ist um einiges kürzer als die erste Version. Der Einsprung für die zweite Programmfunktion (Bildschirm wiederherstellen) ändert sich allerdings ($C00A). Dieser Programmteil wird nun mit SYS 49162 von BASIC aus aufgerufen.

Ein Demoprogramm:

```
100 PRINT CHR$(147):REM BILDSCHIRM LOESCHEN
110 FOR I=1 TO 20
120 : PRINT "TEST DER WINDOWING-ROUTINE"
130 NEXT
140 :
150 SYS 49152:REM BILDSCHIRM RETTEN
160 PRINT CHR$(147):PRINT "BITTE TASTE BETAETIGEN"
170 GET A$:IF A$="" THEN 170:REM AUF TASTE WARTEN
180 SYS 49162:REM BILDSCHIRM ZURUECKHOLEN
```

Das Demoprogramm schreibt zwanzigmal den String "TEST DER WINDOWING-ROUTINE" auf den Bildschirm, ruft die Routine zum Retten auf, löscht den Bildschirm und wartet auf eine Taste. Anschließend wird die Routine zum Zurückholen des Bildschirminhalts aufgerufen und der alte Zustand wiederhergestellt.

C16, Plus/4: Bei diesen beiden Rechnern stellt sich die Frage, wo ein 1024 Byte langer, nicht von BASIC benutzter freier Bereich ist. Die einfachste Lösung: ergänzen Sie das BASIC-Programm um die Zeile:

```
90 GRAPHIC 1,0:GRAPHIC 0
```

Mit diesen Befehlen wird die hochauflösende Grafik ein- und sofort wieder ausgeschaltet. Dabei reserviert der C16, Plus/4 den Bereich $2000-$3FFF für den Grafikspeicher. Diese Reservierung bleibt auch nach dem Ausschalten erhalten.

Ab $2000 stehen uns somit acht Kilobyte zur freien Verfügung. Ändern Sie entsprechend im Assembler-Programm die Einrichtung des Zeigers. Statt auf $9000 sollte er auf $2000 weisen.

Die Einsprungpunkte für den C16 und Plus/4:

| | |
|---|---|
| Bildschirm retten: | SYS 1015 |
| Bildschirm holen: | SYS 1025 |

Wie Sie sehen, kann man bereits mit relativ kleinen Assembler-Programmen äußerst nützliche und vielseitig anwendbare Routinen erstellen.

### 19.1.3. Zugriff auf beliebig große Speicherbereiche

In den vorangegangenen Beispielen wurden jeweils komplette 256-Byte-Blöcke mit der indirekt-indizierten Adressierung behandelt. Es ist jedoch mit ein wenig mehr Aufwand problemlos möglich, auch krumme Blocklängen zu bearbeiten (z.B. 258 Byte = ein 256-Byte-Block plus die ersten beiden Byte des folgenden Blocks).

Das nächste Übungsprogramm invertiert die ersten zehn Zeilen des Bildschirms, behandelt also 400 Byte (ein 256-Byte-Block plus 144 (=$90) Byte des folgenden Blocks).

```
Vorbereitung
A C000 A9 00      LDA #$00          ;LOW-BYTE VON $0400
A C002 85 FB      STA $FB           ;NACH $FB
A C004 A9 04      LDA #$04          ;HIGH-BYTE VON $0400
A C006 85 FC      STA $FC           ;NACH $FC
A C008 A2 01      LDX #$01          ;1 KOMPLETTER BLOCK
A C00A A0 FF      LDY #$FF          ;Y INITIALISIEREN
Innere Schleife
A C00C B1 FB      LDA ($FB),Y       ;AKKU MIT ($FB),Y LADEN
A C00E 18         CLC               ;ADDITION VORBEREITEN
A C00F 69 80      ADC #$80          ;$80 ADDIEREN
A C011 91 FB      STA ($FB),Y       ;UND NACH ($FB),Y ZURÜCK
A C013 88         DEY               ;ZÄHLER DEKREMENTIEREN
A C014 C0 FF      CPY #$FF          ;UND MIT $FF VERGLEICHEN
A C016 D0 F4      BNE $C00C         ;SPRUNG, WENN UNGLEICH
```

```
Äußere Schleife
A C018 E6 FC      INC $FC          ;HIGH-BYTE DES ZEIGERS INKR.
A C01A CA         DEX              ;BLOCKZÄHLER VERRINGERN
A C01B 30 06      BMI $C023        ;SPRUNG, WENN NEGATIV ($FF!)
A C01D D0 ED      BNE $C00C        ;NOCH EINEN BLOCK BEARBEITEN
A C01F A0 8F      LDY #$8F         ;Y MIT RESTBYTES INITIALISIEREN
A C021 D0 E9      BNE $C00C        ;UND ZUR INNEREN SCHLEIFE
A C023 00         BRK              ;FERTIG
```

Daß die Schleifenführung in diesem Programm recht komplex ist, verdanken wir dem Blockrest von 144 Bytes, der nach Bearbeitung eines kompletten Blocks zu behandeln ist.

Zuerst wird wie üblich in $FB/$FC ein Zeiger auf die Startadresse des Bildschirmspeichers ($0400) eingerichtet und das X-Register mit der Anzahl vollständiger Blöcke (mit je 256 Byte) geladen.

Das Y-Register wird nicht wie gewohnt mit $00, sondern mit $FF initialisiert. Warum, sehen wir bei der Behandlung des Restes.

In der inneren Schleife wird nun der Reihe nach auf die Speicherzellen $04FF-$0400 zugegriffen (Y wird dekrementiert). Zu den jeweiligen Bildschirmcodes wird $80 addiert und damit der Code invertiert, bevor er in die betreffende Speicherzelle zurückgeschrieben wird.

Interessant ist nun das Schleifenende. Y wird mit $FF verglichen. Im ersten Durchgang besitzt Y diesen Wert und $04FF wird behandelt. Vor dem Vergleich wird Y jedoch dekrementiert (=> Y=$FE) und BNE verzweigt zum Schleifenanfang, ebenso wie in den folgenden Durchgängen.

Im letzten Durchgang besitzt Y den Wert $00. Die Speicherzelle $0400 wird invertiert, Y wird dekrementiert (=> Y=$FF) und ist somit identisch mit dem Vergleichswert $FF. BNE verzweigt nicht mehr, die innere Schleife wird verlassen.

Wie üblich wird nun das High-Byte des Zeigers erhöht, der anschließend auf den folgenden Block weist (auf $0500). Der Blockzähler X wird dekrementiert.

Der Branch-Befehl BMI interessiert uns momentan noch nicht. Da X den Wert $00 besitzt, wird er jedenfalls nicht ausgeführt (X ist - noch - positiv).

BNE überprüft, ob bereits alle - kompletten - Blöcke behandelt wurden. Da nur ein Block zu bearbeiten war (LDX #$01), ist dies der Fall. X besitzt momentan den Wert $00 und BNE verzweigt nicht.

Nun müssen die restlichen 144 (=$90) Byte des folgenden Blocks behandelt werden. Y wird mit $8F geladen und erneut zum Beginn der inneren Schleife gesprungen. Diese wird wiederum durchlaufen, bis Y den Wert $FF annimmt. Behandelt wird somit der restliche Bereich von $058F bis $0500 ($90 Byte).

Wenn die innere Schleife verlassen wird, erfolgt erneut die Inkrementierung des Zeiger-High-Bytes und die Dekrementierung des Blockzählers X. Das X-Register besaß zuvor den Wert $00 und nimmt nun den negativen Wert $FF an. Die Bedingung BMI ist erfüllt und es wird zum Programmende (BRK) verzweigt.

Ich versprach Ihnen, die Erklärung für den Vergleich CPY #$FF nachzuliefern. Betrachten wir den Fall, daß Y wie üblich mit $00 initialisiert wird und die Verzweigung ohne zusätzlichen Vergleichsbefehl mit BNE erfolgt:

```
(ADRESSE) LDA ($FB),Y
          ...
          ...
          DEY
          BNE (ADRESSE)
          ...
          ...
          LDY #$8F
          BNE (ADRESSE)
```

Mit dieser Schleife werden alle vollständigen Blöcke korrekt behandelt, jedoch nicht der Rest. Nachdem Y mit $8F geladen und zu (ADRESSE) verzweigt wurde, werden der Reihe nach die Speicherzellen $058F-$0501 behandelt. Wenn Y den Wert $01 annimmt, bewirkt die folgende Dekrementierung (DEY), daß die Bedingung BNE nicht erfüllt ist, die Schleife wird verlassen.

Das erste Byte des nur teilweise zu behandelnden letzten Blocks - die Speicherzelle $0500 - wird somit ausgelassen.

Mit dem vorliegenden Programm besitzen Sie eine allgemeine Routine zur Behandlung beliebiger Speicherbereiche. Voraussetzung zum Einsatz dieser Routine ist jedoch, daß die betreffenden Blöcke mit einer rückwärts zählenden Schleife behandelt werden können, was im nächsten Programm nicht der Fall ist.

### 19.1.4. Fehlermeldungen ausgeben

Das folgende Demoprogramm gibt alle Fehlermeldungen des BASIC-Interpreters auf dem Bildschirm aus. Diese Fehlermeldungen befinden sich im ROM (nicht beschreibbarer, nur lesbarer Speicher), und zwar von $A19E-$A327 beim C64, und von $8471-$8652 beim C16, Plus/4. Beim C64 müssen daher 393 Bytes bearbeitet werden, also ein kompletter 256-Byte-Block ($A19E-$A29D) und die ersten 137 (=$89) Zeichen des folgenden Blocks ($A29E-$A327).

Die Fehlermeldungen sind auf eine ganz spezielle Art und Weise abgelegt. Um das Ende einer Fehlermeldung zu erkennen, ist beim letzten Zeichen Bit 7 gesetzt. Da der Code dieses letzten Zeichens daher größer als $79 ist, kann das Ende einer Meldung nach dem Lesen der einzelnen Zeichen mit LDA ($FB),Y durch BMI oder BPL erkannt werden.

Die Meldung selbst ist im ASCII-Code gespeichert und wird von uns mit BSOUT Zeichen für Zeichen ausgegeben.

Da dieses Programm um einiges komplexer, ist als alle bisherigen Demoprogramme, erläutere ich zuerst den prinzipiellen Ablauf:

1. In $FC/FB wird ein Zeiger auf die Anfangsadresse $A19E eingerichtet. Zur Ausgabe der Fehlermeldungen muß eine aufwärts zählende Schleife verwendet werden, sonst wird TOO MANY FILES in der Form SELIF YNAM OOT ausgegeben.

   Diese aufwärts zählende Schleife stellt bei der Behandlung des Restes ein Problem dar. Bei welchem Byte die Behandlung eines Blockes endet, wird der inneren Schleife mit der Speicherzelle $033C mitgeteilt, die zu Beginn den Wert $00 erhält (kompletten Block behandeln).

   Den Rest der Vorbereitung bildet wie üblich das Laden des X-Registers mit der Blockanzahl ($01) und die Initialisierung des Y-Registers mit $00.

2. In der inneren Schleife wird Zeichen für Zeichen ab jener Position gelesen, auf die der Zeiger in $FB/$FC weist. Ist der ASCII-Code kleiner als $80, handelt es sich nicht um das letzte Zeichen einer Fehlermeldung. Das Zeichen wird mit BSOUT ausgegeben und zum Schleifenende verzweigt, um den folgenden Programmteil zu überspringen.

3. Dieser Programmteil führt eine Sonderbehandlung durch, wenn das letzte Zeichen einer Fehlermeldung erkannt wurde. $80 wird vom ASCII-Code subtrahiert, um das Zeichen zu normalisieren und anschließend mit BSOUT auszugeben. Jede Fehlermeldung soll in einer eigenen Zeile ausgegeben werden. Zum Abschluß einer Meldung wird daher der ASCII-Code $0D (=dezimal 13: Code der RETURN-Taste) ausgegeben und ein Zeilenvorschub bewirkt.

4. Bevor die nächste Fehlermeldung ausgegeben wird, wartet das Programm (mit GETIN) auf eine Tastenbetätigung, um ein Durchscrollen des Bildschirms zu vermeiden (beim C64

werden 29 Fehlermeldungen ausgegeben, der Bildschirm umfaßt jedoch nur 25 Zeilen).

5. Y wird inkrementiert und mit dem Inhalt von $033C verglichen. Bei Ungleichheit erfolgt ein weiterer Durchgang, das nächste Zeichen wird gelesen.

6. Ebenso wie im vorigen Programm wird nach der kompletten Ausgabe eines Blocks das High-Byte des Zeigers inkrementiert, der Blockzähler X dekrementiert und - wenn ungleich null - eventuell der nächste Block ausgegeben.

7. Nach vollständiger Ausgabe aller Blöcke erhält $033C (der Vergleichswert) den Wert $8A, da die innere Schleife erneut gestartet wird, nun jedoch nur $90 Byte des letzten Blocks auszugeben sind ($00-$8A).

```
Vorbereitung
A C000 A9 9E      LDA #$9E        ;LOW-BYTE VON $A19E
A C002 85 FB      STA $FB         ;NACH $FB
A C004 A9 A1      LDA #$A1        ;HIGH-BYTE VON $A19E
A C006 85 FC      STA $FC         ;NACH $FC
A C008 A9 00      LDA #$00        ;SCHLEIFENENDWERT $00
A C00A 8D 3C 03   STA $033C       ;NACH $033C (=>KOMPLETTE BLÖCKE)
A C00D A2 01      LDX #$01        ;1 KOMPLETTER BLOCK
A C00F A0 00      LDY #$00        ;Y INITIALISIEREN
Innere Schleife
A C011 B1 FB      LDA ($FB),Y     ;AKKU MIT ($FB),Y LADEN
A C013 30 06      BMI $C01B       ;SPRUNG, WENN GRÖSSER ALS $79
A C015 20 D2 FF   JSR $FFD2       ;ZEICHEN AUSGEBEN (BSOUT)
A C018 4C 33 C0   JMP $C033       ;SONDERBEHANDLUNG ÜBERSPRINGEN!
A C01B 38         SEC             ;SUBTRAKTION VORBEREITEN
A C01C E9 80      SBC #$80        ;$80 SUBTRAHIEREN
A C01E 20 D2 FF   JSR $FFD2       ;NORMALISIERTES ZEICHEN AUSGEBEN
A C021 A9 0D      LDA #$0D        ;$0D (=CODE FÜR RETURN)
A C023 20 D2 FF   JSR $FFD2       ;AUSGEBEN (BSOUT)
A C026 98         TYA             ;Y-WERT AUF
A C027 48         PHA             ;STACK RETTEN
A C028 8A         TXA             ;X-WERT AUF
A C029 48         PHA             ;STACK RETTEN
```

```
A C02A 20 E4 FF     JSR $FFE4        ;MIT GETIN AUF
A C02D F0 FB        BEQ $C02A        ;TASTE WARTEN
A C02F 68           PLA              ;X-WERT VOM
A C030 AA           TAX              ;STACK HOLEN
A C031 68           PLA              ;Y-WERT VOM
A C032 A8           TAY              ;STACK HOLEN
A C033 C8           INY              ;SCHLEIFENZÄHLER DEKREMENTIEREN
A C034 CC 3C 03     CPY $033C        ;MIT $033C VERGLEICHEN
A C037 D0 D8        BNE $C011        ;SPRUNG, WENN UNGLEICH
Äußere Schleife
A C039 E6 FC        INC $FC          ;HIGH-BYTE DES ZEIGERS INKREMENT.
A C03B CA           DEX              ;BLOCKZÄHLER DEKREMENTIEREN
A C03C 30 0A        BMI $C048        ;FERTIG (X NEGATIV)?
A C03E D0 D1        BNE $C011        ;MIT KOMPLETTEN BLOCKS FERTIG?
A C040 A9 8A        LDA #$8A         ;RESTLICHE BYTEANZAHL
A C042 8D 3C 03     STA $033C        ;NACH $033C (=VERGLEICHSWERT)
A C045 4C 11 C0     JMP $C011        ;=> ANFANG INNERE SCHLEIFE
A C048 00           BRK              ;FERTIG !!!
```

Im Grunde genommen besteht kein allzu großer Unterschied zwischen diesem und dem vorhergehenden Programm. Der Schleifenaufbau ist sehr ähnlich, abgesehen davon, daß diese Schleife vorwärts gezählt wird.

$033C erhält zu Beginn den Wert $00 (siehe Vorbereitung). Die innere Schleife beginnt mit dem Y-Wert $00, der bis $FF aufwärts gezählt wird. Beim folgenden Durchgang entspricht der Inhalt des Y-Registers dem Vergleichswert $00 in der Speicherzelle $033C und ein kompletter Block wurde behandelt.

Das High-Byte des Zeigers ($FB/$FC) wird inkrementiert und weist nun auf den nächsten 256-Byte-Block. Der Blockzähler X wird dekrementiert und besitzt - da nur ein Block komplett zu behandeln ist (LDX #$01) - nun den Wert $00. Die Bedingung BNE $C011 ist nicht erfüllt und die restlichen Bytes sollen ausgegeben werden.

$033C erhält nun den Wert $8A, entsprechend der noch ausstehenden Byteanzahl. Mit JMP $C011 wird die innere Schleife erneut gestartet und von Y=$00 bis Y=$8A aufwärts gezählt.

Folgende Besonderheiten sollten Sie beachten:

- Wenn die innere Schleife nach der Behandlung eines kompletten Blocks verlassen wird, besitzt das Y-Register immer (!) den Inhalt $00, da die Behandlung eines vollständigen Blocks bedeutet, daß Y bis zum Vergleichswert $00 (in $033C) inkrementiert wird. Der folgende Sprung zum Schleifenanfang kann daher zu LDA ($FB),Y erfolgen, das Laden des Y-Registers mit $00 (LDY #$00) ist überflüssig.

- Da GETIN ($FFE4) - die Betriebssystem-Routine zur Tastaturabfrage - die Registerinhalte verändert, müssen diese zuvor auf den Stack gerettet und nach Verlassen der Warteschleife wieder vom Stack geholt werden.

- Wie im letzten Programm verzweigt BMI zum Programmende, wenn nach dem letzten vollständigen Block die restlichen Bytes übertragen und X erneut dekrementiert wurde. X besitzt dann den Wert $FF und die Bedingung BMI ("verzweige, wenn negativ") ist erfüllt.

- Gleiches gilt für die Überprüfung des letzten Zeichens einer Fehlermeldung (Bit 7 gesetzt). Auch hier verzweigt BMI, wenn der ASCII-Code des Zeichens größer ist als $79 (negativ = Werte zwischen $80 und $FF).

- Wenn Sie wollen, können Sie den Befehl JMP $C011 durch BNE $C011 ersetzen. Der letzte Befehl, durch den das Zero-Flag beeinflußt wurde, ist LDA #$8A, der zu einem Ergebnis ungleich null führt. Der bedingte Sprung mit BNE wird in diesem speziellen Fall immer (!) ausgeführt.

Für C16, Plus/4-Besitzer: dieses Programm verwendet alle Befehle und Routinen, die wir bisher behandelt haben. Entsprechend viel ist bei der Umsetzung auf Ihren Rechner anzupassen.

Da sich der Beginn der Fehlermeldungen beim C16, Plus/4 bei $8471 befindet, müssen Sie den Zeiger entsprechend korrigieren

(und als Speicherzellen für den Zeiger wie erläutert statt $FB/$FC bei Ihrem Rechner $D8/$D9 verwenden).

Die Fehlermeldungen umfassen beim C16 und Plus/4 den Bereich $8471-$8652, also 481 Byte (einen kompletten Block mit 256 Byte plus die ersten 225 Byte des folgenden Blocks). Der Blockzähler X wird ebenfalls mit $01 geladen, die Anzahl der Restbytes ändern Sie jedoch bitte in $E1 (=dezimal 225).

Vergessen Sie bitte nicht, bei der Eingabe des Programms (ab $03F7) die Sprungadressen (BNE, BMI, JMP) den Adressen der jeweiligen Sprungziele anzupassen.

## 20. Logische Operationen

In den vorangegangenen Kapiteln beschäftigten wir uns ausschließlich mit Bytes, die wir eingelesen haben, speicherten, addierten oder inkrementierten.

Nun bewegen wir uns ein Stockwerk tiefer und steigen auf die Bit-Ebene herab. Mit den Befehlen AND, ORA und EOR können wir gezielt einzelne Bits beeinflussen.

Das Prinzip ist immer gleich: der Akkumulatorinhalt wird mit einem beliebigen Wert nach den Regeln einer der genannten logischen Operationen verknüpft. Das Ergebnis befindet sich ebenso wie bei arithmetischen Operationen anschließend im Akkumulator. Bei jeder logischen Operation werden je zwei Bits miteinander verknüpft. Das Ergebnis besteht wiederum in einem Bit, das entweder gesetzt (1) oder aber gelöscht (0) ist.

### 20.1. Die UND-Verknüpfung

Zuerst beschäftigen wir uns mit der AND-Verknüpfung (UND-Verknüpfung). Zwei Bits, die mit AND verknüpft werden, ergeben nur dann das Resultat 1, wenn beide Bits gesetzt sind.

```
0 AND 0 => Ergebnis 0
0 AND 1 => Ergebnis 0
1 AND 0 => Ergebnis 0
1 AND 1 => Ergebnis 1
```

Nach dieser Regel werden alle acht Bits (Bit 0-7) zweier Bytes miteinander verknüpft. Nehmen wir an, wir laden den Wert $03 in den Akkumulator und verknüpfen ihn mit der Zahl $80.

```
      00000011 (=$03)
AND   10000000 (=$80)
--------------------
      00000000 (=$00)
```

Das Ergebnis ist null, da kein Bit in beiden Zahlen zugleich gesetzt ist. Ein Gegenbeispiel:

```
      11000111 (=$C7)
AND   10011110 (=$9E)
--------------------
      10000110 (=$86)
```

In beiden Zahlen sind die Bits 1,2 und 7 zugleich gesetzt. Nach den Regeln der AND-Verknüpfung werden diese drei Bits auch im Ergebnis gesetzt. Alle anderen Bits werden gelöscht. Im Akkumulator befindet sich nach erfolgter Verknüpfung der Wert $86.

Der Sinn des AND-Befehls besteht darin, gezielt bestimmte Bits einer Zahl zu löschen. Dazu verknüpfen wir die Zahl mit einer sogenannten Maske. Ein Beispiel: wir wollen Bit 6 und Bit 7 einer bestimmten Zahl löschen. Dazu laden wir die betreffende Zahl zuerst in den Akkumulator und verknüpfen sie anschließend mit der Maske 00111111, d.h., mit dem Wert $3F.

```
      00000000        10101010        11111111
AND   00111111  AND   00111111  AND   00111111
------------    ------------    ------------
      00000000        00101010        00111111
```

Die drei Beispiele zeigen, daß die Bits 6 und 7 im Ergebnis immer (!) gelöscht werden, egal welche Zahl mit der Maske verknüpft wird. Alle übrigen Bits bleiben unverändert erhalten.

Schema zum gezielten Löschen von Bits:

1. Das Byte, in dem bestimmte Bits gelöscht werden sollen, wird in den Akkumulator geladen.

2. Der Akkumulator wird mit einer Maske AND-verknüpft, in der die zu löschenden Bits gelöscht und alle übrigen Bits gesetzt sind.

3. Nach der Verknüpfung befindet sich das Ergebnis im Akkumulator. Bis auf die gezielt gelöschten Bits entspricht es exakt dem Ausgangswert.

```
Inverses A auf Bildschirm schreiben und auf Taste warten
A C000 A9 81      LDA #$81        ;BILDSCHIRMCODE A INVERS
A C002 8D 00 04   STA $0400       ;NACH $0400 SCHREIBEN
A C005 20 E4 FF   JSR $FFE4       ;MIT GETIN AUF
A C008 F0 FB      BEQ $C005       ;TASTE WARTEN
Inverses A mit AND-Verknüpfung normalisieren
A C00A AD 00 04   LDA $0400       ;ZEICHEN AN ADRESSE $0400 LESEN
A C00D 29 7F      AND #$7F        ;AND-VERKNÜPFUNG MIT $7F
A C00F 8D 00 04   STA $0400       ;ERGEBNIS ZURÜCKSCHREIBEN
A C012 00         BRK
```

Dieses kleine Programm zeigt eine praktische Anwendung der AND-Verknüpfung. Wie wir wissen, unterscheiden sich die inversen Bildschirm- und ASCII-Codes von den normalen Zeichen durch ein gesetztes siebtes Bit.

$81 (=%10000001) ist der Bildschirmcode für ein inverses A. Dieses Zeichen wird in die obere linke Bildschirmecke geschrieben ($0400, beim C16, Plus/4 $0C00) und anschließend mit GETIN auf Betätigung einer Taste gewartet.

Nach dieser Vorbereitung erfolgt die eigentliche Demonstration. Wenn Sie eine beliebige Taste betätigen, wird das Programm

fortgesetzt. Der Inhalt von Speicherzelle $0400 wird in den Akkumulator gelesen und mit dem Wert $7F (%01111111) ANDverknüpft. Anschließend wird das Ergebnis zurückgeschrieben und auf dem Bildschirm sehen Sie anstelle des inversen A's ein normal dargestelltes A.

```
      10000001 (=$81)   Inverses A
AND   01111111 (=$7F)   Maske zum Löschen von Bit 7
-------------------
      00000001 (=$01)   Normales A
```

## 20.2. Die ODER-Verknüpfung

Die ORA-Verknüpfung (ODER-Verknüpfung) ist das Gegenteil der AND-Verknüpfung. Werden zwei Bits mit AND verknüpft, ist das Ergebnis immer dann ein gesetztes Bit, wenn entweder ein oder aber beide Bits gesetzt waren (inclusives ODER).

```
0 ORA 0 => 0
0 ORA 1 => 1
1 ORA 1 => 1
1 ORA 1 => 1
```

In der Praxis bedeutet diese Eigenschaft der ORA-Verknüpfung, daß wir gezielt Bits setzen können. Wir verknüpfen das betreffende Byte mit einer Maske, in der das gewünschte Bit gesetzt und alle anderen Bits gelöscht sind. Das folgende Schema zeigt, wie mit der Maske %11000000 in beliebigen Zahlen die Bits 6 und 7 gesetzt werden.

```
      00000000          10101010          11111111
ORA   11000000    ORA   11000000    ORA   11000000
-----------------------------------------------------
      11000000          11101010          11111111
```

Schema zum gezielten Setzen von Bits:

1. Wir laden den betreffenden Wert in den Akkumulator.

2. Wir verknüpfen den Akkumulator mit einer Maske, in der alle Bits gelöscht sind, außer jenen, die gesetzt werden sollen.

3. Als Ergebnis erhalten wir im Akkumulator den Originalwert, wobei jedoch alle Bits gesetzt sind, die auch in der Maske gesetzt waren. Alle anderen Bits bleiben unverändert erhalten.

Mit ORA können wir z.B. Zeichen invertieren, ohne $80 addieren zu müssen. Wir setzen einfach das siebte Bit, indem wir den jeweiligen Bildschirmcode mit der Maske %10000000 ($80) verknüpfen:

```
A C000 A2 00      LDX #$00       ;ZÄHLER INITIAL.
A C002 BD 00 04   LDA $0400,X    ;ZEICHEN LESEN
A C005 09 80      ORA #$80       ;ORA-VERKNÜPFUNG MIT $80
A C007 9D 00 04   STA $0400,X    ;ZURÜCKSCHREIBEN
A C00A CA         DEX            ;ZÄHLER DEKREMENT.
A C00B D0 F5      BNE $C002      ;FERTIG?
A C00D 00         BRK
```

Dieses Programm invertiert die ersten 256 Zeichen auf dem Bildschirm. Zeichen für Zeichen wird gelesen und mit der Maske $80 (=%10000000) OR-verknüpft. Bis auf das gesetzte siebte Bit unverändert werden die - invertierten - Zeichen zurückgeschrieben.

## 20.3. Die EXKLUSIV-ODER-Verknüpfung

Mit der AND- und der ORA-Verknüpfung sind wir in der Lage, unser Programm blinkende Bildschirmzeile einfacher zu gestalten. Es geht jedoch noch weitaus komfortabler, wenn wir die EOR-Verknüpfung (Exklusic-ODER) verwenden.

Werden zwei Bits mit EOR verknüpft, resultiert nur dann ein gesetztes Bit, wenn entweder das eine oder (!) das andere Bit gesetzt war.

Beachten Sie bitte die Formulierung entweder oder. Sind beide Bits gesetzt, ist das Ergebnis-Bit im Gegensatz zur ORA-Verknüpfung nicht gesetzt!

```
0 EOR 0 => 0
0 EOR 1 => 1
1 EOR 0 => 1
1 EOR 1 => 0
```

EOR setzt man häufig zur Umkehrung von Bits ein. Wird ein beliebiger Wert mit der Maske %11111111 EOR-verknüpft, kehren sich alle Bitzustände um.

```
      00000000        10101010        11111111
EOR   11111111  EOR   11111111  EOR   11111111
------------    ------------    ------------
      11111111        01010101        00000000
```

Wollen Sie nur ein bestimmtes Bit umdrehen, verknüpfen Sie es (mit EOR) mit einer Maske, in der nur dieses Bit gesetzt und alle anderen gelöscht sind. Beispielsweise können mit der Maske %11000000 die Bits 6 und 7 in beliebigen Zahlen umgedreht werden.

```
      00000000        10101010        11111111
EOR   11000000  EOR   11000000  EOR   11000000
------------    ------------    ------------
      11000000        01101010        00111111
```

Schema zur gezielten Umkehrung von Bits:

1. Laden Sie den betreffenden Wert in den Akkumulator.

2. Verknüpfen Sie den Wert (EOR-Verknüpfung) mit einer Maske, in der nur die umzukehrenden Bits gesetzt und alle

anderen Bits gelöscht sind.

3. Im Akkumulator befindet sich nun der Originalwert, wobei jedoch die in der Maske gesetzten Bits gegenüber dem Original umgedreht sind.

EOR bietet uns daher eine fantastische Möglichkeit, unsere blinkende Bildschirmzeile so einfach wie möglich zu verwirklichen: wir kehren Bit 7 einfach bei jedem Aufruf der Routine um, indem wir die betreffenden Bildschirmcodes mit der Maske %10000000 (=$80) "eoren". Ist Bit 7 gesetzt, wird es gelöscht und umgekehrt.

Um das Ganze etwas abwechslungsreicher zu gestalten, lassen wir im Gegensatz zu früheren Programmen diesmal den kompletten Bildschirm blinken:

```
Blinkender Bildschirm

Vorbereitung
A C000 A9 00      LDA #$00          ;LOW-BYTE VON $0400
A C002 85 FB      STA $FB           ;NACH $FB
A C004 A9 04      LDA #$04          ;HIGH-BYTE VON $0400
A C006 85 FC      STA $FC           ;NACH $FB
A C008 A2 04      LDX #$04          ;BLOCKZÄHLER MIT $04 LADEN
A C00A A0 00      LDY #$00          ;ZÄHLER INNEN INITIALISIEREN
Bildschirm invertieren/normalisieren
A C00C B1 FB      LDA ($FB),Y       ;ZEICHEN LESEN
A C00E 49 80      EOR #$80          ;BIT 7 UMDREHEN
A C010 91 FB      STA ($FB),Y       ;ZEICHEN ZURÜCKSCHREIBEN
A C012 88         DEY               ;ZÄHLER INNEN DEKREMENT.
A C013 D0 F7      BNE $C00C         ;EIN BLOCK FERTIG?
A C015 E6 FC      INC $FC           ;ZEIGER: HIGH-BYTE INKREMENT.
A C017 CA         DEX               ;BLOCKZÄHLER DEKREMENT.
A C018 D0 F2      BNE $C00C         ;ALLE 4 BLOCKS FERTIG?
```

```
Warteschleifen
A C01A 88          DEY             ;ZÄHLER INNEN DEKREMENT.
A C01B D0 FD       BNE $C01A       ;256 DURCHGÄNGE?
A C01D CA          DEX             ;ZÄHLER AUSSEN DEKREMENT.
A C01E D0 FA       BNE $C01A       ;256 DURCHGÄNGE?
A C020 4C 00 C0    JMP $C000       ;NEUSTART DES PROGRAMMS
```

Wenn Sie dieses Programm eingeben, wundern Sie sich eventuell, welche Effekte mit einem derartig kurzen Programm zu erzielen sind. Das Prinzip der Bildschirmbehandlung mit der indirekt-indizierten Adressierung werde ich nicht erläutern, da der verwendete Schleifenaufbau inzwischen hinreichend bekannt sein dürfte.

Die innere Schleife ist dank EOR extrem kurz. Wir müssen nicht mehr überprüfen, ob ein Zeichen invertiert oder normalisiert ist. EOR erledigt die Umkehrung des jeweiligen Zustandes automatisch.

Interessant sind die ebenfalls extrem kurzen, ineinander geschachtelten, Warteschleifen am Programmende (unbedingt nötig, da die Blinkfrequenz sonst viel zu hoch ist).

Diese Warteschleife ist so kurz, da die Tatsache ausgenutzt wird, daß sowohl X als auch Y nach dem Verlassen der Hauptschleife den Wert $00 haben und eine Neuinitialisierung (LDX #$00 und LDY #$00) daher überflüssig ist.

## 20.4. Wahrheitstabellen und Adressierungsarten

Um die Auswirkungen bestimmter logischer Verknüpfungen zu überprüfen, benutzt man normalerweise Wahrheitstabellen, wie sie nachstehend für alle drei Verknüpfungsarten abgebildet sind.

| AND | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| ORA | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| EOR | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 0 |

Um diese Tabellen anzuwenden, prüfen Sie, ob sich am Kreuzungspunkt der jeweiligen Bitzustände eine null oder eine eins befindet.

Beispiel: Sie verknüpfen zwei gesetzte Bits mit EOR. In der EOR-Tabelle finden Sie am Kreuzungspunkt der beiden Bitzustände 1 (=gesetzt) den Wert null vor. Die Bedeutung: wird ein gesetztes Bit mit einem ebenfalls gesetzten Bit EOR-verknüpft, resultiert als Ergebnis ein gelöschtes Bit.

Bei allen drei Verknüpfungen können die folgenden Adressierungsarten verwendet werden:

```
Unmittelbar:          AND #$01
Absolut:              AND $0400
Zeropage:             AND $FF
X-indiziert:          AND $0400,X
Y-indiziert:          AND $0400,Y
Indirekt-indiziert:   AND ($FB),Y
Indiziert-indirekt:   AND ($FB,X)
Zeropage-X-indiziert: AND $FB,X
```

Stören Sie sich bitte nicht an den beiden zuletzt aufgeführten und noch unbekannten Adressierungsarten, die relativ selten verwendet werden.

## 21. Die Schiebe- und Rotierbefehle

Wir sahen, wie Bits gezielt gesetzt, gelöscht oder umgedreht werden. Die Schiebebefehle ASL, LSR, ROL und ROR ermöglichen es uns, das Bitmuster, aus dem ein Byte besteht, beliebig zu verschieben.

## 21.1. ASL (Arithmetisches nach links schieben)

ASL bedeutet arithmetic shift left oder Arithmetisches nach links schieben. Wird keine bestimmte Adresse nach ASL angegeben, bezieht sich der Befehl - ebenso wie bei allen anderen Schiebebefehlen - auf den Inhalt des Akkumulators (Akkumulator-Adressierung).

Alle Bits des im Akkumulator enthaltenen Wertes wandern um eine Stelle nach links. Aus Bit 0 wird Bit 1, aus Bit 1 wird Bit 2 und so weiter. Das freigewordene Bit 0 wird immer (!) mit 0 (gelöschtes Bit) gefüllt.

```
        11110000          01010101          00011011
ASL=>   11100000   ASL=>  10101010   ASL=>  00110110
```

Bit 7 verschwindet bei dieser Schiebung. Es wandert in das neunte Bit des Akkumulators, ins Carry-Bit, und kann mit BCS und BCC getestet werden.

Der Sinn dieser Linksverschiebung? Betrachten Sie bitte den Wert der drei Dualzahlen vor und nach ASL. Bei Dualzahlen bedeutet Verschiebung um eine Stelle nach links Verdopplung. Entsprechend verdoppelt sich der Zahlenwert der drei als Beispiele verwendeten Bitmuster:

```
Ausgangswert:   00000001
ASL       =>    00000010
ASL       =>    00000100
ASL       =>    00001000
ASL       =>    00010000
ASL       =>    00100000
ASL       =>    01000000
ASL       =>    10000000
ASL       =>    00000000
```

Diese Reihe von ASL-Befehlen zeigt, wie das Bitmuster nach links verschoben und von rechts eine null ins Bit 0 nachgeschoben wird. Jede Verschiebung entspricht einer Verdoppelung der Zahl.

Nach dem letzten ASL-Befehl erfolgte ein Übertrag des zuvor gesetzten siebten Bits ins Carry-Bit. Ob das ins Carry-Bit geschobene Bit gesetzt oder aber gelöscht war, können Sie nach einem ASL-Befehl mit BCS ("verzweige, wenn Carry gesetzt") und BCC ("verzweige, wenn Carry gelöscht") abfragen.

Wie erläutert, bezieht sich ASL ohne nähere Angabe auf den Inhalt des Akkumulators. Außer dieser Akkumulator-Adressierung sind folgende Adressierungsarten möglich:

```
Absolut:              ASL $0400
Zeropage:             ASL $FF
X-indiziert:          ASL $0400,X
Zeropage-X-indiziert: ASL $FF,X
```

Angenommen, Sie wollen eine Zahl, die sich in Speicherzelle $033C befindet, mit vier multiplizieren. Dann schieben Sie den Inhalt von $033C zweimal nach links:

```
ASL $033C   ;MAL 2
ASL $033C   ;NOCHMAL MAL 2
```

Komplizierter liegt der Fall jedoch, wenn Sie eine Zahl z.B. mit 21 multiplizieren wollen. In diesem Fall ist ein wenig Nachdenken gefordert.

Die Frage ist, wie man den Faktor 21 am geschicktesten in Binärpotenzen zerlegen kann, also in 1, 2, 4, 8, 16, 32, 64 und 128.

```
21 = 16 + 4 + 1
```

Diese Zerlegung zeigt, daß es möglich ist, die Multiplikation einer Zahl mit 21 zu ersetzen, indem die Zahl zuerst mit 16 (Zahl * 16) und dann mit vier multipliziert wird (Zahl * 4), die Ergebnisse addiert und zum Schluß die Zahl selbst noch einmal addiert wird (Zahl * 1).

Gehen wir von der Zahl $0A aus, die sich im Akkumulator befinden soll:

```
A C000 A9 0A       LDA #$0A        ;AUSGANGSZAHL $0A
A C002 8D 3C 03    STA $033C       ;NACH $033C BRINGEN
A C005 0A          ASL             ;=> ZAHL * 2
A C006 0A          ASL             ;=> ZAHL * 4
A C007 8D 3D 03    STA $033D       ;ZAHL * 4 NACH $033D BRINGEN
A C00A 0A          ASL             ;=> ZAHL * 8
A C00B 0A          ASL             ;=> ZAHL * 16
A C00C 18          CLC             ;ADDITION VORBEREITEN
A C00D 6D 3D 03    ADC $033D       ;ZAHL * 4 ADDIEREN
A C010 6D 3C 03    ADC $033C       ;ZAHL * 1 ADDIEREN
A C013 00          BRK             ;ERGEBNIS: ZAHL * 21 IM AKKU
```

Die Zahl $0A befindet sich im Akkumulator und wird zuerst nach $033C gerettet. Die beiden folgenden ASL-Befehle entsprechen einer Multiplikation mit vier. Das Ergebnis ZAHL*4 kommt nach $033D.

Zwei weitere ASL-Befehle entsprechen wieder einer Multiplikation mit vier. ZAHL*4 mulipliziert mit vier ergibt das zweite Ergebnis, ZAHL*16. Nun wird das erste Ergebnis ZAHL*4 (in $033D) und zum Schluß der Ausgangswert ZAHL (in $033C) addiert.

Nach dem Aufruf des Monitors (BRK) gibt die Registeranzeige den Wert $D2 (=dezimal 210) als Inhalt des Akkumulators aus, das 21fache des Ausgangswerts $0A (=dezimal 10).

## 21.2. LSR (Logisches nach rechts schieben)

LSR (logical shift right, logisches nach rechts schieben) ist die Umkehrung des Befehls ASL.

Während ASL das angegebene Bitmuster (im Akkumulator oder einer Speicherzelle) nach links schiebt, das Bit 7 ins Carry-Bit schiebt und als neues Bit 0 immer ein gelöschtes Bit einfügt, arbeitet LSR umgekehrt. Das Bitmuster wird nach rechts gescho-

ben, wobei Bit 0 herausfällt und ins Carry-Bit kommt. Als neues siebtes Bit wird 0 eingefügt:

```
        11110000            01010101            00011011
LSR=>   01111000    LSR=>   00101010    LSR=>   00001101
```

```
Ausgangswert:   10000000
LSR       =>    01000000
LSR       =>    00100000
LSR       =>    00010000
LSR       =>    00001000
LSR       =>    00000100
LSR       =>    00000010
LSR       =>    00000001
LSR       =>    00000000
```

Die Tabelle zeigt das zugrundeliegende Schema. Jeder ASL-Befehl entspricht einer Division durch zwei. Die ersten sieben Befehle schieben jeweils eine 0 ins Carry-Bit (Carry-Flag gelöscht).

Nach dem achten ASL-Befehl erhalten wir als Ergebnis den Wert null, das zuvor in der Zahl gesetzte Bit 0 befindet sich im Carry-Bit, das nun gesetzt ist. Wie üblich, kann der Zustand des Carry-Bits oder -Flags mit BCS und BCC getestet werden.

## 21.3. ROR und ROL (nach rechts/links rotieren)

ROR (rotate right, rotiere nach links) und ROL (rotate left, rotiere nach rechts) besitzen eine etwas andere Funktionsweise als ASL und LSR.

Beide Befehle führen sogenannte Ringverschiebungen durch. Der Unterschied zu ASL und LSR: in die freiwerdende Stelle beim Links- bzw. Rechtsschieben wird nicht automatisch eine null nachgeschoben, sondern das aktuelle Carry-Bit (null oder eins).

```
CLC=> Carry löschen
Ausgangswert:  10111000
       ROL=>  11100000  (Carry gesetzt)
       ROL=>  11000001  (Carry gesetzt)
```

Der Befehl CLC ist in diesem Beispiel notwendig, um vor einem Rotierbefehl für einen definierten Ausgangszustand des Carry-Bits zu sorgen (gelöscht). Im Beispiel wird das Bitmuster mit ROL nach links geschoben.

Als Bit 0 wird das momentane Carry-Bit (0 wegen CLC) nachgeschoben". Das siebte Bit (1) wandert ins Carry-Bit, das nun gesetzt ist. Der folgende ROL-Befehl schiebt das Bitmuster wieder um eine Stelle nach links, diesmal wird als Bit 0 jedoch eine 1 nachgeschoben, da das Carry-Bit aufgrund des vorangegangenen ROR-Befehls gesetzt ist. Das Carry-Bit wird wiederum gesetzt, da auch dieser ROL-Befehl ein gesetztes siebtes Bit ins Carry-Bit schiebt.

Daraus geht hervor, daß neun ROL-Befehle den ursprünglichen Zustand des Ausgangswerts wiederherstellen (Voraussetzung: vor dem ersten ROL-Befehl wird das Carry-Bit mit CLC gelöscht).

```
Ausgangswert:  11110000
CLC      =>             (Carry gelöscht)
ROL      =>  11100000   (Carry gesetzt)
ROL      =>  11000001   (Carry gesetzt)
ROL      =>  10000011   (Carry gesetzt)
ROL      =>  00000111   (Carry gesetzt)
ROL      =>  00001111   (Carry gelöscht)
ROL      =>  00011110   (Carry gelöscht)
ROL      =>  00111100   (Carry gelöscht)
ROL      =>  01111000   (Carry gelöscht)
ROL      =>  11110000   (Carry gelöscht)
```

ROR arbeitet genau entgegengesetzt. Wie bei LSR findet eine Linksverschiebung statt, in das freigewordene siebte Bit wird nicht wie bei LSR eine null, sondern ebenso wie bei ROL das aktuelle Carry-Bit nachgeschoben:

```
CLC=> Carry löschen
Ausgangswert:  10100111
       ROR=>   01010011  (Carry gesetzt)
       ROR=>   10101001  (Carry gesetzt)
```

Ebenso wie bei ROL kann mit neun Ringverschiebungen der ursprüngliche Zustand wiederhergestellt werden:

```
Ausgangswert:   11110000
CLC       =>            (Carry gelöscht)
ROR       =>   01111000  (Carry gelöscht)
ROR       =>   00111100  (Carry gelöscht)
ROR       =>   00011110  (Carry gelöscht)
ROR       =>   00001111  (Carry gelöscht)
ROR       =>   00000111  (Carry gesetzt)
ROR       =>   10000011  (Carry gesetzt)
ROR       =>   11000001  (Carry gesetzt)
ROR       =>   11100000  (Carry gesetzt)
ROR       =>   11110000  (Carry gesetzt)
```

Alle Schiebe- und Rotierbefehle können mit den bereits bei ASL erläuterten Adressierungsarten eingesetzt werden. Nicht nur der Inhalt des Akkumulators, sondern beliebiger Speicherzellen kann somit geschoben und rotiert werden.

Wie die folgenden Abschnitte zeigen, sind ROL und ROR vor allem in Verbindung mit den Schiebebefehlen ASL und LSR sehr effektiv einzusetzen.

## 21.4. Gezieltes Testen von Bits

Angenommen, wir müssen unbedingt wissen, ob ein bestimmtes Bit in einer Speicherzelle gesetzt oder aber gelöscht ist. Beim siebten Bit ist die einfachste Lösung das Laden des Akkumulators mit der jeweiligen Speicherzelle und das Testen auf positiv oder negativ mit BMI und BPL:

```
LDA $033C      ;WERT LADEN
BMI (ADRESSE)  ;SPRUNG, WENN NEGATIV (BIT 7 GESETZT)
```

Dabei wird jedoch der Inhalt des Akkumulators verändert und muß - wenn benötigt - zuvor auf den Stack gebracht und anschließend wieder geholt werden. Einfacher ist die folgende Möglichkeit:

```
ASL $033C      ;INHALT VON $033C EINE STELLE NACH LINKS SCHIEBEN
BCS (ADRESSE) ;SPRUNG, WENN BIT 7 GESETZT WAR (NUN IM CARRY)
```

Um Bit 0 zu testen, verwenden wir LSR:

```
LSR $033C      ;INHALT VON $033C EINE STELLE NACH RECHTS SCHIEBEN
BCS (ADRESSE) ;SPRUNG, WENN BIT 0 GESETZT WAR (NUN IM CARRY)
```

Die Schiebebefehle ASL und LSR werden wir immer dann verwenden, wenn der ursprüngliche Inhalt der Speicherzelle nicht mehr benötigt wird.

Wollen wir hingegen gezielt Bits testen, den Wert selbst jedoch wiederherstellen, wenn der Test negativ ausfiel, verwenden wir (zuvor Carry-Bit mit CLC löschen!) eine Kombination aus ASL bzw. LSR, und ROR bzw. ROL:

```
ASL $033C      ;INHALT VON $033C EINE STELLE NACH LINKS SCHIEBEN
BCS (ADRESSE) ;SPRUNG, WENN BIT 7 GESETZT WAR (NUN IM CARRY)
ROR $033C      ;WIEDER NACH RECHTS SCHIEBEN INCLUSIVE CARRY
```

ASL schiebt das Bitmuster eine Stelle nach links, wobei das siebte Bit im Carry-Bit landet. Nach dem Test schieben wir den Wert mit ROR wieder eine Stelle nach rechts zurück. Der alte Zustand wird wiederhergestellt, da ROR nicht einfach eine null, sondern das Carry-Bit in die freiwerdende Stelle schiebt. Und im Carry-Bit befindet sich ja nach ASL das ursprüngliche siebte Bit des betreffenden Wertes.

Um Bit 0 zu testen, verwenden wir die umgekehrte Kombination (LSR und ROL):

```
LSR $033C      ;INHALT VON $033C EINE STELLE NACH RECHTS SCHIEBEN
BCS (ADRESSE) ;SPRUNG, WENN BIT 7 GESETZT WAR (NUN IM CARRY)
ROL $033C      ;WIEDER NACH LINKS SCHIEBEN INCLUSIVE CARRY
```

Merken Sie sich: sollen nur Bits getestet werden, ist die einfachste Möglichkeit die Verwendung von LSR und ASL. Benötigen wir den Wert jedoch anschließend unverändert, schieben wir das jeweilige Bit mit ASL oder LSR ins Carry-Bit, testen mit BCS oder BCC dieses Bit und stellen anschließend den ursprünglichen Wert mit ROR oder ROL wieder her.

Um andere Bits (0,1,..,6,7) zu testen, schieben wir so lange, bis sich das gewünschte Bit im Carry-Bit befindet und testen dieses mit BCC oder BCS. Soll anschließend der ursprüngliche Wert wiederhergestellt werden, verwenden wir entsprechend viele Rotierbefehle. Beachten Sie jedoch unbedingt, daß nur die erste Schiebung mit ASL oder LSR durchgeführt werden darf, alle folgenden mit ROL oder ROR, da sonst Bits unwiederbringlich verlorengehen.

```
Bit 5 testen

        ASL $033C              ;BIT 7 INS CARRY
        ROL $033C              ;BIT 6 INS CARRY
        ROL $033C              ;BIT 5 INS CARRY
        BCS (ADRESSE)          ;BIT 5 TESTEN
        ROR $033C              ;NACH RECHTS INCLUSIVE CARRY
        ROR $033C              ;NACH RECHTS INCLUSIVE CARRY
        ROR $033C              ;NACH RECHTS INCLUSIVE CARRY
```

Angenommen, in $033C befindet sich der Wert %10111001 (=$B9). Dann ergibt sich folgender Ablauf:

Ausgangszustand: $033C=%10111001, Carry: unbekannt

```
Befehl    $033C        Carry
ASL       01110010     GESETZT
ROL       11100101     GELÖSCHT
ROL       11001010     GESETZT
BCS         KEINE VERÄNDERUNG
ROR       11100101     GELÖSCHT
ROR       01110010     GESETZT
ROR       10111001     GELÖSCHT
```

## 21.5. Weitere Einsatzmöglichkeiten

Wir wissen nun, wie wir mit den Schiebebefehlen einen 8-Bit-Wert mit beliebigen Potenzen zur Basis zwei (2^1=2, 2^2=4, 2^3=8,...) multiplizieren können. Wir schieben ihn ein-, zwei-, dreimal oder auch öfter nach links. Wenn nötig, speichern wir Zwischenergebnisse und können mit deren Hilfe auch ungerade Multiplikationen durchführen.

Wenn wir jedoch eine 16-Bit-Zahl auf diese Weise mit einem Wert multiplizieren wollen, bekommen wir Probleme. Gehen wir von der Zahl $1082 aus, die wir im Adreßformat (zuerst Low-, dann High-Byte) in $033C und $033D ablegen:

```
$033D (High-Byte $10)      $033C (Low-Byte $82)

     %00001010                   %10000010
```

Theoretisch wäre es nun möglich, zuerst das Low- und anschließend das High-Byte nach links zu schieben. Leider bleibt dabei der Übertrag unberücksichtigt, der beim Linksschieben des Low-Bytes entsteht und ins Carry-Bit wandert. Dieser muß im High-Byte anschließend mitberücksichtigt und zu dessem Bit 0 werden:

```
$033D (High-Byte)      $033C (Low-Byte)

 %00010100    <= 1 <=    %00000100
```

Das Problem wird durch den kombinierten Einsatz von ASL und ROL gelöst. Mit ASL werden die acht Bits des Low-Bytes ($033C) nach links verschoben, wobei Bit 7 ins Carry-Bit wandert.

Anschließend wird das High-Byte nicht mit ASL, sondern mit ROL ebenfalls um eine Stelle nach links geschoben. Dabei wird das Carry-Bit - das im Beispiel gesetzt ist - zum Bit 0 des High-Bytes. Ein eventueller Übertrag im Carry-Bit wird durch die Anwendung von ROL auf das High-Byte automatisch berücksichtigt:

```
ASL $033C ;LOW-BYTE * 2
ROL $033D ;HIGH-BYTE * 2 (UNTER BERÜCKSICHTIGUNG EINES ÜBERTRAGS)
```

## 22. Interruptprogrammierung (SEI, CLI, RTI und BRK)

Interrupt-Programme werden oft als die Krönung der Assembler-Programmierung bezeichnet. Mit keiner anderen Programmiertechnik lassen sich derart effektvolle Programme erstellen, deren Auswirkungen dem Laien oftmals völlig unerklärlich sind.

Bevor wir uns diesem Leckerbissen zuwenden, muß ich jedoch die Frage klären, was unter einem Interrupt zu verstehen ist. Im Grunde genommen nichts anderes als eine Unterbrechung.

Der Rechner unterbricht seine laufende Arbeit (z.B. die Ausführung eines BASIC-Programms) für einen Moment, um sich anderen Beschäftigungen zuzuwenden.

Vielleicht fragen Sie sich manchmal, wieso im Direkt-Modus der Cursor ständig blinkt, welches Heinzelmännchen im Rechner ununterbrochen eine Bildschirm-Speicherzelle abwechselnd invertiert und wieder normalisiert.

Oder wer ständig die rechnerinterne Uhr (die Sie in BASIC mit den Variablen TI und TI$ abfragen können) weiterzählt und in jedem Moment die Tastatur abfragt?

Nun, für alle diese grundlegenden Aufgaben sind sogenannte Interrupt-Programme zuständig. Ebenso wie im täglichen Leben (das Telefon klingelt und unterbricht Ihre augenblickliche Tätigkeit) werden Interrupt-Programme immer dann aufgerufen, wenn eine Unterbrechung auftritt.

Die CPU kennt nun verschiedene Unterbrechungsarten und demzufolge verschiedene Interrupts. Die für unsere Zwecke wichtigsten Interrupts heißen IRQ und NMI.

## 22.1. IRQ und NMI

Der IRQ (interrupt request, Unterbrechungsanforderung) besitzt eine andere "Priorität" als die zweite Interrupt-Sorte, der NMI (non maskable interrupt, nicht maskierbarer Interrupt).

Um zu verstehen, was unter Priorät gemeint ist, greife ich wieder auf ein alltägliches Beispiel zurück. Angenommen, Ihre momentane Tätigkeit besteht im Lesen dieses Buches. Nun kommt ein IRQ, eine maskierbare Unterbrechunganforderung. Und zwar klingelt das Telefon.

Sie werden Ihre Tätigkeit - das Lesen - wahrscheinlich unterbrechen und ans Telefon gehen. Nun tritt jedoch ein NMI auf, eine nicht maskierbare Unterbrechung, Sie hören, daß in der Küche das Wasser überkocht.

Der NMI (Wasser kocht) besitzt zweifellos eine höhere Priorität als der IRQ (Telefon klingelt). Die Unterbrechung wird ihrerseits durch eine wichtigere Unterbrechung unterbrochen und Sie werden sofort in die Küche rennen.

Wenn der NMI erledigt ist (Sie nahmen den Topf von der Herdplatte) können Sie sich wieder dem IRQ zuwenden, der weniger wichtigen Tätigkeit des Telefonierens.

Erst wenn auch diese Unterbrechung erledigt ist, wenden Sie sich wieder Ihrer normalen - und sehr sinnvollen - Beschäftigung zu, dieses Buch zu lesen.

Dieses Beispiel beschreibt hervorragend die Reaktion der CPU auf eine Unterbrechungsanforderung. Die Frage ist nun, wie eine solche Unterbrechung erzeugt, wie ein Interrupt ausgelöst wird.

Unser Rechner besitzt die verschiedensten Interruptquellen, vor allem die sogenannten CIA-Bausteine, deren Register unter anderem für die Farben des Bildschirmrahmens (Adresse $D020 beim C64, $FF19 beim C16, Plus/4) und -hintergrundes zuständig sind.

Uns interessiert nun vor allem eine der verschiedenen Unterbrechungsanforderungen, die dieser Baustein erzeugt. Er enthält eine interne Uhr, für die spezielle Register zuständig sind, die sogenannten Timer.

Nach jeder sechzigstel Sekunde lösen diese Timer einen IRQ aus, einen maskierbaren Interrupt. Eine weitere IRQ-Quelle ist der BRK-Befehl. Dieser Befehl löst ebenfalls einen IRQ aus, wobei zugleich das sogenannte BRK-Flag oder Break-Flag im Status-Register (Bit 3) gesetzt wird, damit beim folgenden Ablauf zwischen einem durch die Timer und einem durch einen BRK-Befehl ausgelösten IRQ unterschieden werden kann.

Der Prozessor reagiert auf einen IRQ folgendermaßen:

1. Der Inhalt des Programmzählers - der ja die Adresse des nächsten zu bearbeitenden Befehls enthält - und des Status-Registers werden auf den Stack gerettet.

2. Im Status-Register wird das Bit 1 gesetzt. Solange dieses Bit gesetzt ist, werden keine weiteren IRQ's bearbeitet (nur ein NMI kann einen IRQ unterbrechen).

3. Am Speicherende ($FFFE/$FFFF) befindet sich eine Adresse in der üblichen Form Low-Byte/High-Byte im ROM, also im nicht veränderbaren Speicherteil. Der Programmzähler wird mit dieser Adresse geladen. Die weitere Programmbearbeitung erfolgt daher ab der jeweiligen Adresse.

Punkt 3 entspricht dem bereits bekannten Sprung über einen Vektor (JMP (VEKTOR)). Die Sprungadresse befindet sich unveränderbar in den Speicherzellen $FFFE/$FFFF.

Beim C64 befindet sich in diesen Speicherzellen die Adresse $FF48, die Startadresse einer Routine, die folgende Aufgaben besitzt:

1. Die aktuellen Registerinhalte werden auf den Stack gerettet.

2. Anhand des Status-Registers, das auf den Stack gebracht wurde (siehe oben), wird geprüft, ob der IRQ durch die Timer oder durch einen BRK-Befehl ausgelöst wurde.

3. Bei einem BRK-IRQ erfolgt ein Sprung über den Vektor $0316/$0317 (JMP ($0316)).

4. Bei einem Timer-IRQ erfolgt ebenfalls ein indirekter Sprung, jedoch über den Vektor $0314/$0315, der normalerweise die Adresse $EA31 enthält (JMP ($0314)).

Punkt 3 erklärt den Rücksprung in den Monitor mit dem BRK-Befehl. Jeder Monitor verstellt nach dem Laden und Starten zunächst einmal den BRK-Vektor an Adresse $0316/$0317. Dieser Vektor wird so verändert, daß er anschließend mitten in das Monitor-Programm weist, genauer: auf die Registeranzeige.

Wenn wir mit einem BRK-Befehl unser Programm beenden, wird beim Erreichen dieses Befehls ein IRQ ausgelöst und anschließend über den BRK-Vektor zur Registeranzeige des Monitors gesprungen. Ein Geheimnis wäre damit gelüftet.

Interessanter für unsere Zwecke ist jedoch der Timer-IRQ und der resultierende Sprung über den Vektor $0314/$0315 zur Adresse $EA31 (C64). Ab Adresse $EA31 befindet sich beim C64 die sogenannte Service-Routine, die folgende Aufgaben besitzt:

1. Die Uhr wird weitergezählt.

2. Der Cursor wird invertiert, wenn er gerade normalisiert ist bzw. normalisiert, wenn er momentan invertiert ist.

3. Die Tastatur wird abgefragt, vor allem die STOP-Taste (nun wissen Sie, warum Sie ein BASIC-Programm in jedem Augenblick mit STOP unterbrechen können).

4. Die ursprünglichen Inhalte der Register und des Programmzählers werden wieder vom Stack geholt.

5. Mit dem Befehl RTI (return from interrupt, beende die Unterbrechung) wird dem Prozessor angezeigt, daß die Unterbrechung beendet ist.

Der Prozessor setzt die Bearbeitung des unterbrochenen Programms mit dem nächsten Befehl (der alte Inhalt des Programmzählers wurde wieder vom Stack geholt) fort, als hätte die Unterbrechung niemals stattgefunden.

Aus diesen Erläuterungen geht hervor, wie wir eigene Programme in den Systeminterrupt - der jede sechzigstel Sekunde stattfindet - einbinden können. Wir verändern den IRQ-Vektor $0314/$0315, so daß er statt auf die normale Service-Routine ab $EA31 auf die Startadresse unseres eigenen Programms weist, das jede sechzigstel Sekunde aufgerufen werden soll.

Eines ist dabei jedoch zu beachten: zu jeder ordnungsgemäßen Einbindung einer Interrupt-Routine gehört als letzter Programmbefehl ein Sprung zur Service-Routine (JMP $EA31). Erfolgt dieser Sprung nicht, wird weder die STOP-Taste abge-

fragt, noch die Uhr weitergezählt oder der Cursor kontinuierlich invertiert und normalisiert (=> der Cursor blinkt).

Alle grundlegenden Systemfunktionen sind ausgeschaltet, wenn wir die Service-Routine einfach umgehen. Daher werden eigene Interrupt-Routinen in Form einer Verlängerung eingebunden.

Der IRQ-Vektor wird auf unser Programm verstellt, dieses ausgeführt, und zum Schluß die normale Service-Routine aufgerufen.

Für C16, Plus/4-Besitzer: wie Sie sich bereits denken können, dürfen Sie einmal mehr andere Adressen verwenden. Der IRQ-Vektor befindet sich beim C16 und Plus/4 nicht an der Adresse $0314/$0315, sondern bei $0312/$0313. Die Service-Routine beginnt ebenfalls an einer anderen Adresse. Statt an $EA31, beginnt sie beim C16, Plus/4 an Adresse $CE42.

Angenommen, unsere Interrupt-Routine beginnt an Adresse $C00D. Dann verstellen wir den IRQ-Vektor auf diese Adresse mit den Befehlen:

```
LDA #$0D   ;LOW-BYTE VON $C00D
STA $0314  ;NACH $0314
LDA #$C0   ;HIGH-BYTE VON $C00D
STA $0315  ;NACH $0315
BRK        ;ENDE DER INITIALISIERUNG
```

Falsch!!!

Zwar nicht allzu oft, jedoch ab und zu wird der Rechner abstürzen, wenn wir so vorgehen. Theoretisch könnte genau dann ein Timer-IRQ stattfinden, wenn wir das Low-Byte des Vektors geändert haben.

Da das High-Byte noch unverändert ist ($EA), weist der IRQ-Vektor auf die Adresse $EA0D. Der indirekte Sprung beim Timer-IRQ führt somit zu dieser Adresse. Was sich dort befindet, wissen wir jedoch nicht. Höchstwahrscheinlich kein allzu sinnvolles Programm.

Bevor wir diesen Vektor verändern, müssen wir daher sicherstellen, daß kein IRQ erfolgen kann. Den IRQ kann man (im Gegensatz zum NMI) ausschalten, und zwar mit dem Befehl SEI (set interrupt mask, setze die Interrupt-Maske).

Das Interrupt-Flag des Status-Registers (Bit 2) wird auf 1 gesetzt, was bedeutet, daß folgende Timer- oder BRK-IRQ's nicht ausgeführt werden. Wir haben den IRQ gesperrt und können den IRQ-Vektor verändern, ohne befürchten zu müssen, daß genau in diesem Moment ein Interrupt erfolgt (zumindest kein IRQ, höchstens ein NMI, der jedoch über einen anderen Vektor springt und daher nicht zum Absturz führt).

```
SEI         ;IRQ VERHINDERN
LDA #$0D    ;LOW-BYTE VON $C00D
STA $0314   ;NACH $0314
LDA #$C0    ;HIGH-BYTE VON $C00D
STA $0315   ;NACH $0315
CLI         ;IRQ WIEDER ZULASSEN
BRK
```

Nach dem Verbiegen des Vektors wird mit dem Befehl CLI (clear interrupt mask, Unterbrechungsmaske löschen) das Interrupt-Flag gelöscht und damit IRQ's wieder zugelassen.

Nun wissen wir auch, warum die zweite Interrupt-Sorte, der NMI, eine höhere Priorität besitzt als der IRQ: weil ein NMI nicht verhindert werden kann (NMI = non (!) maskable interrupt).

Doch zurück zu unserer Interrupt-Routine. Beim nächsten IRQ wird nun zur Adresse $C00C gesprungen. Dort befindet sich jedoch noch kein Programm.

Also überlegen wir uns, welche Programmfunktion jede sechzigstel Sekunde ausgeführt werden soll. Zum Aufwärmen zuerst etwas sehr einfaches: jede sechzigstel Sekunde wird das Zeichen A in die obere linke Bildschirmecke geschrieben. Geben Sie bitte ein:

Die erste Interrupt-Routine (C64)

```
IRQ-Vektor "verbiegen"
A C000 78          SEI          ;IRQ SPERREN
A C001 A9 0D       LDA #$0D     ;LOW-BYTE VON $C00D
A C003 8D 14 03    STA $0314    ;NACH IRQ-VEKTOR LOW
A C006 A9 C0       LDA #$C0     ;HIGH-BYTE VON $C00D
A C008 8D 15 03    STA $0315    ;NACH IRQ-VEKTOR HIGH
A C00B 58          CLI          ;IRQ ZULASSEN
A C00C 00          BRK          ;ENDE
Interrupt-Routine
A C00D A9 01       LDA #$01     ;ASCII-CODE VON 'A'
A C00F 8D 00 04    STA $0400    ;IN OBERE LINKE ECKE
A C012 4C 31 EA    JMP $EA31    ;=>SERVICE-ROUTINE
```

Die eigentliche Interrupt-Routine beginnt ab $C00D, unmittelbar hinter der Initialisierung, die den IRQ-Vektor auf die Adresse der Routine verbiegt.

Die Interrupt-Routine gibt ein A in der oberen linken Bildschirmecke aus ($0400) und springt anschließend zur normalen Service-Routine an Adresse $EA31.

Um das Programm zu starten, rufen Sie die Initialisierungs-Routine mit G C000 auf. Die Interrupt-Routine ist nun eingebunden und wird automatisch jede sechzigstel Sekunde ausgeführt.

Das Resultat: das Zeichen A befindet sich immer (!) in der ersten Spalte der ersten Bildschirmzeile, selbst wenn Sie versuchen, es zu überschreiben oder mit der Taste CLR den kompletten Bildschirm löschen.

Der Grund: da das A jede sechzigstel Sekunde - also in extrem kurzen Abständen - ausgegeben wird, ist es für das menschliche Auge sofort nach dem Löschen wieder vorhanden.

Bereits mit diesem wohl denkbar anspruchlosen Interrupt-Programm lassen sich Uneingeweihte verblüffen, denen Sie die Aufgabe stellen, das A zu löschen.

Um die Interrupt-Routine wieder auszuschalten, drücken Sie gleichzeitig die Tasten STOP und RESTORE, mit denen eine Betriebssystem-Routine aufgerufen wird, die verschiedene Zeiger (unter anderem den IRQ-Vektor) neu einrichtet und somit unseren Eingriff rückgängig macht.

```
Die erste Interrupt-Routine (C16, Plus/4)

      IRQ-Vektor "verbiegen"
      A 03F7 78          SEI           ;IRQ SPERREN
      A 03F8 A9 04       LDA #$04      ;LOW-BYTE VON $0404
      A 03FA 8D 12 03    STA $0312     ;NACH IRQ-VEKTOR LOW
      A 03FD A9 04       LDA #$04      ;HIGH-BYTE VON $0404
      A 03FF 8D 13 03    STA $0313     ;NACH IRQ-VEKTOR HIGH
      A 0402 58          CLI           ;IRQ ZULASSEN
      A 0403 00          BRK           ;ENDE
      Interrupt-Routine
      A 0404 A9 01       LDA #$01      ;ASCII-CODE VON 'A'
      A 0406 8D 00 0C    STA $0C00     ;IN OBERE LINKE ECKE
      A 0409 4C 24 CE    JMP $CE42     ;=>SERVICE-ROUTINE
```

Denken Sie bei Interrupt-Routinen für den C16, Plus/4 immer daran, daß sich die Service-Routine bei diesen Rechnern bei $CE42 befindet!

Der NMI wurde bisher vernachlässigt. Diese Unterbrechungsanforderung ist für uns in der Praxis weniger interessant als der IRQ, der von den Timern automatisch jede sechzigstel Sekunde ausgelöst wird.

Eine NMI-Quelle ist z.B. die RESTORE-Taste. Diese Taste löst bei jeder Betätigung einen NMI aus. In den NMI können Sie eigene Routinen einbinden, indem Sie den NMI-Vektor verbiegen ($0318/$0319 beim C64, $0314/$0315 beim C16, Plus/4). Am Ende Ihrer Routine muß sich wiederum ein Sprung zur norma-

len NMI-Routine befinden, zu $8E60 beim C64 bzw. zu $CE0E beim C16, Plus/4.

Mit Hilfe des NMI können Sie jederzeit (!) eine eigene Routine aufrufen, auf die der NMI-Vektor zeigt.

## 22.2. Übungsprogramme zur Interrupt-Programmierung

Für die folgenden Übungsprogramme verwenden wir zwei (!) Routinen zum Verbiegen des IRQ-Vektors. Eine Routine, die unsere Interrupt-Routine einschaltet, und eine zweite, die sie ausschaltet, indem sie den ursprünglichen Inhalt des IRQ-Vektors wiederherstellt.

Um beide Routinen von BASIC aus mit SYS aufrufen zu können, enden Sie nicht mit BRK, sondern mit RTS.

```
Ein-/Ausschalten von Interrupt-Routinen (C64)

      Interrupt-Routine einschalten
      A C000 78         SEI          ;IRQ SPERREN
      A C001 A9 1A      LDA #$1A     ;VEKTOR IN
      A C003 8D 14 03   STA $0314    ;$0314/$0315
      A C006 A9 C0      LDA #$C0     ;AUF
      A C008 8D 15 03   STA $0315    ;$C01A VERBIEGEN
      A C00B 58         CLI          ;IRQ ZULASSEN
      A C00C 60         RTS
      Interrupt-Routine ausschalten
      A C00D 78         SEI          ;IRQ SPERREN
      A C00E A9 31      LDA #$31     ;VEKTOR IN
      A C010 8D 14 03   STA $0314    ;$0314/$0315
      A C013 A9 EA      LDA #$EA     ;WIEDER AUF
      A C015 8D 15 03   STA $0315    ;$EA31 (SERVICE-ROUTINE)
      A C018 58         CLI          ;IRQ ZULASSEN
      A C019 60         RTS
      Interrupt-Routine
      A C01A
```

Mit SYS 49152 ($C000) wird nun die ab $C01A beginnende Interrupt-Routine eingeschaltet und mit SYS 49165 ($C00D) wieder ausgeschaltet.

```
Ein-/Ausschalten von Interrupt-Routinen (C16, Plus/4)

      Interrupt-Routine einschalten
      A 03F7 78            SEI            ;IRQ SPERREN
      A 03F8 A9 11         LDA #$11       ;VEKTOR IN
      A 03FA 8D 12 03      STA $0312      ;$0312/$0313
      A 03FD A9 04         LDA #$04       ;AUF $0411
      A 03FF 8D 13 03      STA $0313      ;VERBIEGEN
      A 0402 58            CLI            ;IRQ ZULASSEN
      A 0403 60            RTS
      Interrupt-Routine ausschalten
      A 0404 78            SEI            ;IRQ SPERREN
      A 0405 A9 42         LDA #$42       ;VEKTOR IN
      A 0407 8D 12 03      STA $0312      ;$0312/$0313
      A 040A A9 CE         LDA #$CE       ;WIEDER AUF
      A 040C 8D 13 03      STA $0313      ;$CE42 (SERVICE-ROUTINE)
      A 040F 58            CLI            ;IRQ ZULASSEN
      A 0410 60            RTS
      Interrupt-Routine
      A 0411
```

C16, Plus/4-Besitzer schalten die Interrupt-Routine (ab $0411) mit SYS 1015 ein und mit SYS 1028 wieder aus.

### 22.2.1. Blinkende Bildschirmzeile

Was hätten Sie den nun gerne jede sechzigstel Sekunde ausgeführt? Wie wär's - um gemütlich zu beginnen - mit der altbekannten blinkenden Bildschirmzeile, diesmal jedoch als Interrupt-Routine und damit um einiges effektvoller?

Dieses Programm demonstriert ein typisches Problem aller Interrupt-Routinen. Die Invertierung/Normalisierung wird wie üblich mit einer Schleife und dem EOR-Befehl vorgenommen:

```
          LDX #$27
(ADRESSE) LDA $0400,X
          EOR #$80
          STA $0400,X
          DEX
          BPL (ADRESSE)
          JMP $EA31
```

Obwohl es nur jede sechzigstel Sekunde aufgerufen wird, ist dieses Programm noch viel zu schnell. Die Bildschirmzeile blinkt nicht, sie flimmert.

Bei einem normalen Programm würden wir dieses Problem mit einer Warteschleife lösen, nicht jedoch bei einer Interrupt-Routine!

Interrupt-Routinen dürfen nicht zu lang sein (wichtig ist die Zeitdauer, nicht die Programmlänge). Sonst wird noch während des Interrupts der nächste Timer-IRQ ausgelöst, und das dann Unfug entstehen muß (!), ist einsichtig.

Also müssen wir unser Geschwindigkeitsproblem auf andere Weise lösen, am besten so, wie es uns die Service-Routine beim Blinken des Cursors vormacht. Der Cursorzustand (invertiert/normalisiert) wird nicht bei jedem IRQ umgedreht, sondern nur bei jedem zwanzigsten.

Wir verwenden eine Speicherzelle als Zähler und zwar $033C. Bei jedem Aufruf unserer Interrupt-Routine wird $033C dekrementiert. Ist das Ergebnis ungleich null, überspringen wir die Schleife und springen sofort zur Service-Routine nach $EA31.

Ist das Ergebnis hingegen null, initialisieren wir die Speicherzelle mit dem Wert 20 (=$14) und durchlaufen die Blinkschleife.

IRQ-Routine Blinkende Bildschirmzeile

```
A C01A CE 3C 03     DEC $033C     ;$033C DEKREMENTIEREN
A C01D D0 12        BNE $C031     ;SPRUNG, WENN UNGLEICH NULL
A C01F A9 14        LDA #$14      ;SONST MIT $14
A C021 8D 3C 03     STA $033C     ;NEU INITIALISIEREN
A C024 A2 27        LDX #$27      ;UND DIE OBERSTE
A C026 BD 00 04     LDA $0400,X   ;BILDSCHIRMZEILE MIT $80
A C029 49 80        EOR #$80      ;'EOREN', DAS HEISST
A C02B 9D 00 04     STA $0400,X   ;'UMDREHEN' (INVERTIEREN
A C02E CA           DEX           ;BZW.NORMALISIEREN)
A C02F 10 F5        BPL $C026
A C031 4C 31 EA     JMP $EA31     ;=> SERVICE-ROUTINE
```

Geben Sie das Programm ab Adresse $C01A ein, im Anschluß an den Initialisierungsteil.

C16, Plus/4-Besitzer verwenden bitte entsprechend die Startadresse $0411 und ändern die Startadresse des Bildschirmspeichers ($0400 beim C64) wie üblich in $0C00. Vergessen Sie bitte nicht, die Adresse $EA31 (Service-Routine beim C64) durch $CE42 (Service-Routine beim C16, Plus/4) zu ersetzen und auch die Zieladressen der Sprungbefehle zu korrigieren.

Nach der Einbindung in den Systeminterrupt mit SYS 49152 (C16, Plus/4: SYS 1015) vergehen eventuell bis zu vier Sekunden, bis die Routine loslegt, da der Ausgangswert des Zählers $033C beim ersten Durchgang unbestimmt ist.

Enthält er den Maximalwert $FF, wird die Bildschirmzeile erst nach dem 255ten IRQ invertiert (255*1/60 sec = 4.25 sec). Danach jedoch blinkt die Zeile im gleichen zeitlichen Rhythmus wie der Cursor.

Wenn Sie wollen, können Sie die Initialisierungs-Routine ändern, so daß in dieser der Zähler $033C mit $01 geladen wird. Dann erfolgt die erste Invertierung unmittelbar nach dem Aufruf.

Die geradezu märchenhaften Eigenschaften von Interrupt-Routinen werden deutlich, wenn Sie nun z.B. ein BASIC-Programm

schreiben und starten. Dies alles stört die blinkende Bildschirmzeile nicht im geringsten.

Während im Hintergrund ein Interrupt-Programm läuft, können Sie im Vordergrund ganz normal mit dem Rechner arbeiten. Für den ahnungslosen Benutzer hat es den Anschein, als würden zwei Programe gleichzeitig (!) bearbeitet werden.

Sie wissen nun jedoch, daß das Hintergrund-Programm tatsächlich nur jede sechzigstel Sekunde im Vorbeigehen bearbeitet wird. Ich kenne keine andere Methode, die die unglaubliche Geschwindigkeit von Assembler so eindrucksvoll dokumentiert wie eine Interrupt-Routine.

### 22.2.2. Belegung der Funktionstasten

Dieses zweite Übungsprogramm wendet sich ausschließlich an C64-Besitzer. Beim C16, Plus/4 und C128 können die Funktionstasten mit beliebigen Zeichenketten belegt werden. Bei Betätigung einer Funktionstaste wird sofort die zugehörige Zeichenkette ab der aktuellen Cursorposition ausgegeben. Diese Funktionstastenbelegung ist sehr nützlich, da ein Tastendruck genügt, um häufig benötigte Befehle wie LOAD, SAVE, RUN oder LIST einzugeben.

Mit einem geeigneten Interrupt-Programm können wir dem C64 die gleiche Fähigkeit verleihen. Die Interrupt-Routine wird bei jedem Timer-IRQ aufgerufen und fragt die Tastatur ab. Wurde eine Funktionstaste betätigt, wird die zugehörige Zeichenkette ausgegeben.

Bevor wir mit der Programmerstellung beginnen, sollte jedoch noch ein Problem gelöst werden: Angenommen, wir schreiben ein BASIC-Programm, das die Funktionstasten zum Auslösen bestimmter Funktionen benutzt, z.B.:

```
100 GET A$:IF A$="" THEN GOTO 100:REM WARTESCHLEIFE
110 IF A$=CHR$(133) THEN GOTO 500:REM F1 GEDRUECKT
110 IF A$=CHR$(134) THEN GOTO 500:REM F2 GEDRUECKT
...
...
...
```

Die Funktionstasten werden beim C64 innerhalb eines Programms benutzt, indem die ihnen zugeordneten ASCII-Codes 133 bis 140 wie im Beispiel abgefragt werden. Die im Hintergrund lauernde Interrupt-Routine schreibt wie erläutert bei jeder Betätigung einer Funktionstaste die zugeordnete Zeichenkette auf den Bildschirm, was beim Ablauf eines BASIC-Programms äußerst störend wäre.

Um die normale Funktion dieser Tasten nicht zu beeinflussen, wird im folgenden Programm immer eine Tastenkombination (!) abgefragt. Die Interrupt-Routine wird nur dann aktiv, wenn gleichzeitig mit einer Funktionstaste die Commodore-Taste betätigt wird. Die einfache Betätigung einer Funktionstaste läuft daher weiterhin ab wie gewohnt, d.h., der ASCII-Code der gedrückten Taste wird übergeben (GET A$), weitere Aktionen erfolgen nicht.

Auf diese Weise wird ausgeschlossen, daß die Interrupt-Routine ein BASIC-Programm stört, das die Funktionstasten wie gezeigt zur Steuerung des Programmablaufs verwendet.

Die zu erstellende Routine ist leider recht komplex. Zur Tastaturabfrage können wir nicht einfach wie gewohnt die Routine GETIN verwenden. Auch BSOUT zur Ausgabe der jeweiligen Zeichenkette ist nicht zulässig. Beide Betriebssystem-Routinen sind zu langsam, um in einer Interrupt-Routine verwendet zu werden. Selbst wenn wir GETIN benutzen könnten, wären wir dadurch noch nicht in der Lage, festzustellen, ob der Benutzer gleichzeitig die Commodore-Taste betätigt.

Die Tastaturabfrage lösen wir wie folgt:

1. Abfrage der Commodore-Taste: Der Inhalt der Speicherzelle $028D gibt an, ob momentan eine der Sondertasten SHIFT, CTRL oder die Commodore-Taste gedrückt ist. Wenn die Commmdore-Taste gedrückt wird, enthält $028D den Wert $02 (bei SHIFT den Wert $01 und bei CTRL den Wert $04).

   Wird unsere IRQ-Routine aufgerufen, vergleichen wir somit $028D mit $02. Enthält $028D einen anderen Wert, ist die Commodore-Taste nicht gedrückt und es wird zur normalen Service-Routine verzweigt.

2. Abfrage der Funktionstasten: Wenn die Commodore-Taste betätigt wird, prüfen wir, ob gleichzeitig auch eine Funktionstaste gedrückt wird. Die Speicherzelle $C5 gibt uns über die momentan gedrückte Taste Auskunft. Der Inhalt $03 entspricht der Taste F7, $04 der Taste F1, $05 der Taste F3 und $06 der Funktionstaste F5.

Wie 2. zeigt, beschränken wir uns auf die Abfrage - und Belegung - von vier Funktionstasten, F1, F3, F5 und F7.

```
Tastaturabfrage
A C01A AD 8D 02     LDA $028D      ;'CTRL-FLAG' HOLEN
A C01D C9 02        CMP #$02       ;INHALT $02 = CTRL GEDRUECKT
A C01F D0 30        BNE $C051      ;SPRUNG, WENN NICHT GEDRUECKT
A C021 A5 C5        LDA $C5        ;SONST TASTE HOLEN UND PRUEFEN,
A C023 C9 03        CMP #$03       ;OB ES SICH UM EINE FUNKTIONS-
A C025 90 2A        BCC $C051      ;TASTE HANDELT ($03, $04, $05
A C027 C9 07        CMP #$07       ;ODER $06)
A C029 B0 26        BCS $C051      ;SPRUNG, WENN KEINE FUNKTIONSTASTE
A C02B CD 57 C0     CMP $C057      ;MIT LETZTER TASTE VERGLEICHEN
A C02E F0 21        BEQ $C051      ;SPRUNG, WENN GLEICH
```

Die Tastaturabfrage verzweigt in verschiedenen Fällen zu $C051, wo sich der Befehl JMP $EA31 befindet, also der Einsprung in die Service-Routine des Betriebssystems. Die Verzweigung er-

folgt, wenn $028D nicht den Wert $02 enthält, also CTRL nicht betätigt wird.

Sie erfolgt ebenfalls, wenn $C5 einen Wert enthält, der kleiner ist als $03 oder größer als $07. In beiden Fällen wird keine der vier Funktionstasten betätigt.

Die beiden letzten Befehle (CMP $C057 : BEQ $C051) können Sie nicht verstehen, ohne zu wissen, das am Ende unserer Routine der Inhalt von $C5 - die momentan betätigte Taste - in die Speicherzelle $C057 kopiert wird. In $C057 befindet sich somit bei jedem Aufruf der Code jener Taste, die während des letzten Aufrufs gedrückt wurde.

Ist die momentan gedrückte Taste mit der zuletzt betätigten identisch, wird ebenfalls sofort zur Service-Routine verzweigt, obwohl momentan sowohl eine Funktions- als auch die Commodore-Taste betätigt wird.

Der Grund: Wie wir wissen, wird eine IRQ-Routine jede sechzigstel Sekunde aufgerufen. Da eine Taste normalerweise jedoch länger betätigt wird, wird während einer Tastenbetätigung unsere Routine mehrmals aufgerufen und gibt die zugehörige Zeichenkette ebenfalls mehrmals aus.

Daher überprüfen wir diesen Spezialfall - die beim vorigen Aufruf gedrückte Taste ist mit der momentan gedrückten identisch - und übergehen die Ausgabe der Zeichenkette, wenn wir wissen, daß diese offenbar bereits beim letzten Aufruf ausgegeben wurde.

Der folgende Programmteil soll die auszugebenden Zeichen ermitteln. Die Zeichenketten selbst befinden sich im ASCII-Format am Programmende und beginnen an Adresse $C058. Jede Zeichenkette wird von der vorhergehenden durch die Endemarke $00 getrennt.

Der zweite Programmteil liest diese Zeichenketten ein, bis die Startposition der auszugebenden Zeichen gefunden wurde. Um den folgenden Programmteil zu verstehen, müssen Sie wissen,

daß im Funktionstastenspeicher am Programmende als erstes die der Taste F7 zugeordnete Zeichenkette abgelegt ist und dieser folgend die den Tasten F1, F3 und F5 zugeordneten Zeichenketten.

```
Position der Zeichenkette ermitteln
A C030 A0 00      LDY #$00      ;STARTWERT FUER Y-REGISTER
A C032 A2 03      LDX #$03      ;STARTWERT FUER X-REGISTER
A C034 E4 C5      CPX $C5       ;X MIT ZEICHENCODE VERGLEICHEN
A C036 F0 09      BEQ $C041     ;SPRUNG, WENN GLEICH
A C038 C8         INY           ;SONST Y ERHOEHEN
A C039 B9 58 C0   LDA $C058,Y   ;ZEICHEN LESEN
A C03C D0 FA      BNE $C038     ;UND WEITERLESEN, WENN NICHT $00
A C03E E8         INX           ;WENN $00 GELESEN, X INKREMENTIEREN
A C03F D0 F3      BNE $C034     ;UND VERZWEIGEN
```

Bei diesem zweiten Programmabschnitt handelt es sich um eine relativ schwer zu durchschauende Schachtelung zweier Schleifen. Zu Beginn der äußeren Schleife wird Y mit $00 und X mit $03 (also dem Code der Funktionstaste F7) geladen.

X wird mit dem Inhalt von $C5 verglichen. Wird Gleichheit festgestellt - enthält also $C5 den Wert $03 - wird verzweigt. Die Zeichenketten am Programmende werden nicht weiter durchsucht, da sich die Zeichenkette, die der Taste F7 zugeordnet ist, am Anfang des Funktionstastenspeichers befindet.

Ansonsten wird der Funktionstastenspeicher bis zum Ende der aktuellen Zeichenkette abgesucht, bis zur Endemarke $00. Wird diese Endemarke erreicht, ist die Bedingung BNE ("verzweige, wenn ungleich null") nicht erfüllt und die innere Schleife, die Zeichen für Zeichen liest, wird verlassen.

Der folgende INX-Befehl inkrementiert das X-Register, bevor zum Beginn der äußeren Schleife verzweigt wird. X enthält nun den Wert $04, der dem Code entspricht, mit dem die Taste F1 in der Speicherzelle $C5 dargestellt wird. Das Y-Register wurde bei jedem Durchgang der inneren Schleife inkrementiert und weist nun auf den Beginn der zweiten Zeichenkette, genauer: Auf das

Trennbyte $00, das sich vor dem ersten Zeichen der zweiten Zeichenkette befindet.

Wenn der Inhalt von $C5 mit dem X-Wert $04 identisch ist, wird die Suchschleife verlassen und nach $C041 verzweigt, ansonsten wird die momentan untersuchte Zeichenkette wiederum bis zu ihrem Ende gelesen und unser Funktionstastenzähler X inkrementiert.

Diese sehr komplexe Routine führt letztendlich zu folgendem Ergebnis: Y weist auf das Trennbyte $00, das sich vor jener Zeichenkette befindet, die der gedrückten Funktionstaste zugeordnet ist.

Der dritte Programmteil gibt diese Zeichenkette auf dem Bildschirm aus. BSOUT können wir wie erläutert nicht verwenden. Direkt in den Bildschirmspeicher zu schreiben, bereitet ebenfalls Probleme, da wir zuvor die aktuelle Cursorposition ermitteln müßten.

Eine recht trickreiche Lösung besteht darin, die Zeichenkette in den Tastaturpuffer des C64 zu schreiben. In diesem Tastaturpuffer speichert unser Rechner alle noch nicht verarbeiteten Zeichen. Also tun wir so, als wären die auszugebenden Zeichen zwar vom Benutzer eingetippt, jedoch noch nicht verarbeitet und auf dem Bildschirm ausgegeben worden.

Der Tastaturpuffer beginnt beim C64 ab Adresse $0277. Der folgende Programmteil liest ab der durch den Inhalt des Y-Registers angegebenen Position in unserem Funktionstastenspeicher Zeichen für Zeichen ein und schreibt sie in den Tastaturpuffer.

```
Zeichenkette in Tastaturpuffer übertragen
A C041 A2 00      LDX #$00      ;STARTWERT 1 FUER DAS
A C043 E8         INX           ;X-REGISTER
A C044 C8         INY           ;Y AUF 1.ZEICHEN HINTER $00
A C045 B9 58 C0   LDA $C058,Y   ;ZEICHEN LESEN UND IN DEN
A C048 9D 76 02   STA $0276,X   ;TASTATURPUFFER SCHREIBEN
A C04B D0 F6      BNE $C043     ;FERTIG, WENN $00 GELESEN
```

Der letzte Programmteil teilt dem Betriebssystem mit, daß sich im Tastaturpuffer 'X' noch auszugebende Zeichen befinden. Die Speicherzelle $C6 gibt an, wie viele noch zu bearbeitende Zeichen der Tastaturpuffer enthält. In diese Speicherzelle wird der aktuelle X-Wert übertragen.

Wie bereits erwähnt wurde, wird zuletzt der in $C5 enthaltene Tastencode für den nächsten Aufruf der IRQ-Routine nach $C057 kopiert, bevor mit JMP $EA31 die normale Service-Routine aufgerufen wird.

```
Programm beenden
A C04D 86 C6      STX $C6      ;ANZAHL ZEICHEN IM TAST.PUFFER
A C04F A5 C5      LDA $C5      ;AKTUELLEN TASTENCODE NACH
A C051 8D 57 C0   STA $C057    ;$C057 KOPIEREN
A C054 4C 31 EA   JMP $EA31    ;=> SERVICE-ROUTINE
```

Das Programm selbst ist beendet, es fehlen jedoch noch die Zeichenketten, mit denen die Funktionstasten belegt sind. Diese beginnen ab Adresse $C058 oder dezimal 49240.

Verwenden Sie zur Ablage der Zeichenketten das folgende BASIC-Programm:

```
100 A$=CHR$(0)+"RUN"+CHR$(0)+"LIST"+CHR$(0)+"LOAD"+CHR$(0)+"SAVE"
    + CHR$(0)
110 FOR I=1 TO LEN(A$)
120 : POKE I+49239,ASC(MID$(A$,I,1))
130 NEXT
```

Dieses Programm belegt F7 mit der Zeichenkette RUN, F1 mit LIST, F3 mit LOAD und F5 mit SAVE. Die Belegung können Sie selbstverständlich ändern. Dabei müssen Sie jedoch berücksichtigen, daß der Tastaturpuffer nur zehn Zeichen umfaßt und daher keine Zeichenkette länger als zehn Zeichen sein darf.

Nach dem Starten des BASIC-Programms rufen Sie bitte wieder den Monitor auf und speichern Sie den Bereich $C000 bis

$C0A0 unter dem Namen F-KEYS ab. Außer dem eigentlichen Programm wird auch die Funktionstastenbelegung gespeichert.

Bei einer späteren Verwendung der Routine laden Sie das Assembler-Programm und initialisieren Sie die Interrupt-Routine mit SYS 49152. Mit SYS 49165 kann die Belegung jederzeit ausgeschaltet werden.

Denken Sie bitte daran, daß es nicht ausreicht, eine der vier Funktionstasten zu betätigen. Gleichzeitig müssen Sie die Commodore-Taste drücken.

Bevor Sie das folgende, vollständige Programm-Listing eingeben, erlauben Sie mir noch eine Bemerkung. Dieses Programm ist zweifellos erheblich schwerer zu verstehen als alle vorangegangenen Demoprogramme. Pädagogisch gesehen gehört es zweifellos nicht in ein Lehrbuch, das mit den Grundlagen der Maschinensprache beginnt.

Der Sinn dieses Programmes besteht vor allem darin, Sie zu eigenen Experimenten mit Interrupt-Routinen zu motivieren. Das letzte Demoprogramm bewirkte das Blinken einer Bildschirmzeile. Dieses Programm war problemlos zu verstehen und zeigte die bei der Interrupt-Programmierung zu beachtenden Prinzipien auf. Eine sinnvolle Anwendung war es jedoch nicht, sondern fiel eher unter die Rubrik Spielerei.

In diese Abteilung fällt ein großer Teil der in diversen Fachzeitschriften veröffentlichten IRQ-Routinen, z.B. die immer wiederkehrenden Laufschrift-Programme.

Das Demoprogramm Funktionstastenbelegung sollte Ihnen zeigen, daß mit ein wenig Phantasie äußerst nützliche IRQ-Routinen vorstellbar sind. Das vorgestellte Programm erleichtert die BASIC-Programmierung derart, daß Sie sich schnell angewöhnen werden, es nach jedem Einschalten des Rechners als erstes Programm zu laden.

Noch ein Tip: Da dieses Programm vor allem bei der BASIC-Programmierung verwendet wird und daher nur in seltenen Fäl-

len zuvor ein Monitor geladen wurde, müssen Sie die Routine von BASIC aus laden.

Geben Sie nach dem Einschalten des Rechners (oder wann immer Sie die Routine laden wollen) ein: LOAD"F-KEYS",8,1 und nach dem Beenden des Ladevorgangs NEW. Mit diesen beiden Befehlen wird der LOAD-Befehl des Monitors ersetzt, das Programm kann von BASIC aus geladen werden (vor dem BASIC-Programm, das durch NEW gelöscht würde).

Nachfolgend finden Sie das Listing des kompletten Programms. Nicht abgedruckt ist die im vorigen Kapitel erläuterte Initialisierungs-Routine, die ab $C000 einzugeben ist.

Das vollständige Programm-Listing:

```
Tastaturabfrage
A C01A AD 8D 02     LDA $028D       ;'CTRL-FLAG' HOLEN
A C01D C9 02        CMP #$02        ;INHALT $02 = CTRL GEDRUECKT
A C01F D0 30        BNE $C051       ;SPRUNG, WENN NICHT GEDRUECKT
A C021 A5 C5        LDA $C5         ;SONST TASTE HOLEN UND PRUEFEN,
A C023 C9 03        CMP #$03        ;OB ES SICH UM EINE FUNKTIONS-
A C025 90 2A        BCC $C051       ;TASTE HANDELT ($03, $04, $05
A C027 C9 07        CMP #$07        ;ODER $06)
A C029 B0 26        BCS $C051       ;SPRUNG, WENN KEINE FUNKTIONSTASTE
A C02B CD 57 C0     CMP $C057       ;MIT LETZTER TASTE VERGLEICHEN
A C02E F0 21        BEQ $C051       ;SPRUNG, WENN GLEICH
Position der Zeichenkette ermitteln
A C030 A0 00        LDY #$00        ;STARTWERT FUER Y-REGISTER
A C032 A2 03        LDX #$03        ;STARTWERT FUER X-REGISTER
A C034 E4 C5        CPX $C5         ;X MIT ZEICHENCODE VERGLEICHEN
A C036 F0 09        BEQ $C041       ;SPRUNG, WENN GLEICH
A C038 C8           INY             ;SONST Y ERHOEHEN
A C039 B9 58 C0     LDA $C058,Y     ;ZEICHEN LESEN
A C03C D0 FA        BNE $C038       ;UND WEITERLESEN, WENN NICHT $00
A C03E E8           INX             ;WENN $00 GELESEN, X INKREMENTIEREN
A C03F D0 F3        BNE $C034       ;UND VERZWEIGEN
```

```
Zeichenkette in Tastaturpuffer übertragen
A C041 A2 00      LDX #$00       ;STARTWERT 1 FUER DAS
A C043 E8         INX            ;X-REGISTER
A C044 C8         INY            ;Y AUF 1.ZEICHEN HINTER $00
A C045 B9 58 C0   LDA $C058,Y    ;ZEICHEN LESEN UND IN DEN
A C048 9D 76 02   STA $0276,X    ;TASTATURPUFFER SCHREIBEN
A C04B D0 F6      BNE $C043      ;FERTIG, WENN $00 GELESEN
Programm beenden
A C04D 86 C6      STX $C6        ;ANZAHL ZEICHEN IM TAST.PUFFER
A C04F A5 C5      LDA $C5        ;AKTUELLEN TASTENCODE NACH
A C051 8D 57 C0   STA $C057      ;$C057 KOPIEREN
A C054 4C 31 EA   JMP $EA31      ;=> SERVICE-ROUTINE
```

# Teil 3: Eigenheiten verschiedener Rechner - Programmierhilfsmittel

Im vorangegangenen Fortgeschrittenen-Kursus wurden Ihnen derart viele Befehle und Adressierungsarten vorgestellt, daß Sie eine kleine Pause verdient haben.

Auf den folgenden Seiten löse ich daher mein Versprechen ein und beschreibe Ihnen die Eigenheiten Ihres C16, Plus/4 oder C128. Ich empfehle jedem Besitzer eines C128, sich auch die auf den C16, Plus/4 bezogenen Erläuterungen durchzulesen, da der C128 seinen Speicher auf prinzipiell gleiche Art und Weise verwaltet (allerdings ein wenig komplizierter).

Anschließend zeige ich Ihnen, welche Hilfsmittel uns zur Eingabe und zum Testen größerer Assembler-Programme zur Verfügung stehen.

## 23. C16, C116 und Plus/4

Besitzer eines C16, C116 oder Plus/4 sahen bereits in den vorangegangenen Kapiteln, daß sich die Assembler-Programmierung Ihres Rechners nicht wesentlich von der des C64 unterscheidet.

Vor allem zwei Unterschiede sind zu berücksichtigen:

1. Unterschiedliche Adressen des Bildschirmspeichers, der Zeropage-Speicherzellen und freier Speicherbereiche, die für Assembler-Programme genutzt werden können.

2. Das Bankswitching, mit dem der Plus/4 (und der erweiterte C16, C116) den RAM-Bereich unter dem ROM zur Ablage von Daten nutzt.

## 23.1. Freie Speicherbereiche

Zwei Unterschiede zum C64 kennen Sie bereits: die Startadresse des Bildschirmspeichers ($0C00) und den bisher für Assembler-Programme verwendeten freien Speicherbereich $03F7-$0436.

Die folgende Tabelle zeigt, welche Speicherbereiche meines Wissens nach beim C16, C116 und Plus/4 unbenutzt sind und zur Programm- und Datenspeicherung verwendet werden können.

Beachten Sie bitte, daß die angeführten Bereiche in den seltensten Fällen völlig ungenutzt sind. Es gibt mehrere Bereiche, die nur in speziellen Fällen vom Betriebssystem genutzt werden. Ein Beispiel ist der Bereich $03F7-$0436, den wir zur Programmablage verwendeten.

Das Betriebssystem benutzt diesen Bereich nur beim - seltenen - Betrieb der RS-232-Schnittstelle (z.B. zum Betrieb eines Modems). Wird diese Schnittstelle nicht benötigt, können Sie in diesem Bereich nach Belieben ohne jede Überschreibgefahr Daten oder Programme speichern.

Weitere Bereiche benötigt der BASIC-Interpreter. Diese Bereiche können Sie ebenfalls nach Belieben nutzen, wenn Sie Assembler-Programme schreiben, die weder die - später beschriebenen - Routinen des BASIC-Interpreters verwenden noch mit einem BASIC-Programm zusammenarbeiten sollen.

Sind diese Bedingungen nicht erfüllt, dürfen diese Bereiche nur zur vorübergehenden Datenablage verwendet werden, nicht jedoch zur Programmspeicherung. Wenn das Assembler-Programm beendet ist (=> Rückkehr nach BASIC) oder eine Routine des BASIC-Interpreters aufruft, besteht für diese Bereiche Überschreibgefahr!

Im folgenden führe ich nur jene Speicherbereiche auf, die bereits von mir selbst praktisch erprobt wurden. Weitere nutzbare Bereiche zu entdecken überlasse ich Ihnen.

*Die Fließkomma-Akkumulatoren ($61-$70)*

Den Bereich $61-$70 in der Zeropage benötigt der BASIC-Interpreter für arithmetische Berechnungen. Wenn Sie reine Assembler-Programme schreiben, können Sie in diesem Bereich unbesorgt Daten und das Programm selbst ablegen.

Verwenden Sie - wie in den folgenden Kapiteln beschrieben - Assembler-Routinen, die zusammen mit BASIC-Programmen arbeiten sollen, dürfen Sie diesen Bereich nur zur vorübergehenden Datenspeicherung verwenden, jedoch nicht zur Ablage von Assembler-Programmen, da der BASIC-Interpreter diesen Bereich - und ein eventuell darin enthaltenes Programm - bei der nächsten Bearbeitung einer BASIC-Programmzeile oder einer Berechnung überschreibt.

*Der reservierte Zeropage-Bereich ($D8-$E8)*

Der Bereich $D8-$E8 ist speziell für Anwendungs-Software - also unsere Assembler-Programme - reserviert und wird weder vom BASIC-Interpreter noch vom Betriebssystem benutzt.

In diesem Bereich können Sie nach Belieben Daten dauerhaft (ohne Überschreibgefahr) ablegen.

*Der Graphik-Cursor ($02AD-$02CB)*

Den Bereich $02AD-$02AD benutzt der BASIC-Interpreter nur bei eingeschalteter hochauflösender Graphik. In reinen Assembler-Programmen kann dieser Bereich ohne Bedenken genutzt werden.

Sie können diesen Bereich auch dann verwenden, wenn Sie zwar Routinen zur Zusammenarbeit mit einem BASIC-Programm schreiben, dieses jedoch ausschließlich im Textmodus arbeitet.

*Der RS-232-Input-Puffer ($03F7-$0436)*

Dieser 64 Byte lange Bereich bietet sich zur Speicherung kurzer Assembler-Programme an. Solange die RS-232-Schnittstelle nicht betrieben wird, ist er dauerhaft vor dem Überschreiben geschützt.

*Der Funktionstasten-Speicher ($055F-$05E6)*

Wie Sie wissen, können Sie die Funktionstasten beim C16 und Plus/4 mit beliebigen Strings belegen. Die Strings werden im Bereich $055F-$05E6 abgelegt. Wenn Sie diese Tasten nicht benötigen, steht Ihnen somit ein 135 Byte langer Bereich für Programme oder umfangreiche Datenmengen zur Verfügung.

Jedoch Vorsicht: der Bereich wird überschrieben, wenn Sie mit dem KEY-Befehl einer Funktionstaste einen String zuweisen.

*Der Sprachsynthesizer-Bereich ($065E-$06EB)*

Der Bereich $065E-$06EB wird für den Sprachsynthesizer verwendet. Wenn Sie darauf verzichten können, stehen Ihnen weitere 141 Byte uneingeschränkt zur Verfügung.

**Große Assembler-Programme**

Die bisher aufgeführten Bereiche eignen sich vorwiegend zur Datenablage oder Speicherung kleinerer Assembler-Programme. Wollen Sie jedoch komplette Textverarbeitungen oder Dateiverwaltungen in Assembler schreiben, benötigen Sie weit größere Speicherbereiche für Ihr Programm.

Zwei Möglichkeiten bieten sich an:

- Bei reinen Assembler-Programmen steht Ihnen der komplette BASIC-Speicher zur Verfügung. Dieser Bereich erstreckt sich beim C16, C116 von $1000-$3FFF, wenn die hochauflösende Graphik nicht eingeschaltet ist, ansonsten von $1000-$17FF.

  Beim größeren Plus/4-Speicher - und beim auf 64 KByte erweiterten C16, C116 - erstreckt sich das BASIC-RAM von $1000-$FCFF bzw. - wenn die hochauflösende Graphik eingeschaltet ist - von $4000-$FCFF.

- Wenn die hochauflösende Graphik eingeschaltet ist, wird der Bereich $1C00-$1FFF für die sogenannte Luminanztabelle und der Bereich $2000-$3FFF für den Graphik-Bildschirm reserviert.

  Wenn Sie große Assembler-Programme schreiben, die mit BASIC-Programmen zusammenarbeiten sollen, bietet es sich an, dem BASIC-Programm diesen Speicher wegzuschnappen. Schalten Sie mit GRAPHIC 1,0 die hochauflösende Graphik ein und legen Sie Ihr Assembler-Programm in den nun reservierten Bereich $1C00-$3FFF.

## 23.2. Das Bankswitching

Wenn Sie einen C16, C116 mit 16 KByte RAM besitzen (Grundausführung) wird Sie dieses Kapitel nicht im geringsten interessieren, da Sie niemals mit dem Bankswitching in Berührung kommen.

Besitzer eines Plus/4 oder eines auf 64 KByte ausgebauten C16, C116 sollten jedoch wissen, daß sich im 64 KByte-Adreßbereich ($0000-$FFFF) Ihres Rechners weit mehr Speicherzellen als nur 64 KByte befinden.

Sie verfügen über 64 KByte an RAM-Speicher und zusätzlichen ROM-Speicher, in dem sich das Betriebssystem, der BASIC-In-

terpreter, der Monitor und - beim Plus/4 - die eingebaute Software befindet.

Diese ROM-Bereiche überdecken das darunterliegende RAM im Normalfall. Es ist für den Prozessor nicht sichtbar und kann daher auch nicht ohne Probleme angesprochen werden.

Um genau zu sein: Schreibzugriffe (STA $FFFF) sprechen immer (!) den RAM-Bereich an, da ROM-Speicherzellen bekanntlich nicht beschrieben werden können.

Wollen Sie nun eine RAM-Speicherzelle auslesen, die vom RAM verdeckt ist, werden Sie sich wundern. Anstelle des zuvor z.B. mit STA $FFFF nach $FFFF geschriebenen Wertes liest der Befehl LDA $FFFF einen völlig anderen Wert aus der Speicherzelle $FFFF.

Der Prozessor sieht beim Lesen nicht das verdeckte RAM, sondern ausschließlich den darüberliegenden ROM-Bereich. Der Lesezugriff LDA $FFFF liest daher den Inhalt einer Speicherzelle des im ROM fest eingebauten Betriebssystem-Programms.

Dieser problematische überlagerte RAM-Bereich beginnt sowohl beim Plus/4 als auch beim auf 64 KByte ausgebauten C16, C116 ab Adresse $8000.

Glücklicherweise können Sie jedoch jederzeit die Speicherkonfiguration umschalten und wählen, ob der Prozessor im Bereich $8000-$FFFF den ROM- oder aber den RAM-Bereich sehen soll.

Mit einem beliebigen (!) Schreibbefehl in eines von zwei Registern des TED-Bausteins schalten Sie zwischen ROM und RAM um:

Umschaltung auf RAM: `STA $FF3F`
Umschaltung auf ROM: `STA $FF3E`

Diese beiden Speicherzellen können immer (!) angesprochen werden, unabhängig von der momentan eingeschalteten Konfiguration:

```
Lesen und Schreiben ohne Konfigurations-Umschaltung

A 03F7 A9 01        LDA #$01        ;$01 NACH
A 03F9 8D 00 F0     STA $F000       ;$F000 SCHREIBEN
A 03FC AD 00 F0     LDA $F000       ;INHALT VON $F000 LESEN
A 03FF 00           BRK
```

Wenn Sie dieses Programm starten, stellen Sie anschließend anhand der Registeranzeige fest, daß nicht der Wert $01, sondern $05 in den Akkumulator geladen wurde. Das Programm schreibt in den verdeckten RAM-Bereich, liest jedoch aus dem darüberliegenden ROM-Bereich.

```
Lesen und Schreiben mit Konfigurations-Umschaltung

A 03F7 A9 01        LDA #$01        ;$01 NACH
A 03F9 8D 00 F0     STA $F000       ;$F000 SCHREIBEN
A 03FC 8D 3F FF     STA $FF3F       ;RAM EINSCHALTEN
A 03FF AD 00 F0     LDA $F000       ;INHALT VON $F000 LESEN
A 0402 8D 3E FF     STA $FF3E       ;ROM EINSCHALTEN
A 0405 00           BRK
```

Nach Ablauf dieses Programms zeigt die Registeranzeige den ert $01 als Inhalt des Akkumulators an. Der Wert wurde ins RAM geschrieben (Schreibzugriffe erfolgen immer auf das RAM!) und vor dem Lesezugriff wurde die RAM-Konfiguration geändert. Der ROM-Bereich wurde aus- und der RAM-Bereich eingeblendet (STA $FF3F, beliebiger Schreinzugriff auf $FF3F). Daher wird korrekt aus der RAM-Speicherzelle $F000 der Wert $01 gelesen.

Sofort nach dem Lesezugriff wird die alte Konfiguration wiederhergestellt. Der Schreibzugriff auf $FF3E blendet das RAM wieder aus und das ROM ein. Diese Herstellung der Stan-

dardkonfiguration sollte aus Sicherheitsgründen immer erfolgen, wenn Ihre Lesezugriffe beendet sind.

Der Grund: wenn auf RAM geschaltet ist, sieht der Prozessor die im ROM festeingebauten Programme nicht. Ein Befehl wie JSR $FFD2 (BSOUT) zu einer Betriebssystem-Routine führt daher ins RAM. Der Inhalt der betreffenden RAM-Speicherzellen ist jedoch undefiniert, völlig zufällig. Mit an Sicherheit grenzender Wahrscheinlichkeit wird sich der Rechner aufhängen.

## 24. C128

Den C128 im 128er-Modus zu programmieren, ist leider ein düsteres Kapitel, gespickt mit Schwierigkeiten aller Art. Das erste Problem stellt sich bereits bei der Suche nach einem freien Speicherbereich für das Programm ein.

Obwohl Ihr Rechner 128 KByte RAM besitzt, werden Sie sehen, daß es gar nicht einfach ist, selbst für kleinere Assembler-Programme freie Speicherbereiche zur Ablage zu finden.

Ein weiteres Problem stellt der im 80-Zeichen-Modus nicht direkt adressierbare Bildschirmspeicher dar. Die folgenden Seiten beschreiben diese Probleme und bieten Ihnen Lösungen an.

### 24.1. Freie Speicherbereiche

Der C128 besitzt 128 KByte RAM, der Prozessor kann jedoch nur maximal 64 KByte ($0000-$FFFF) adressieren, also ansprechen. Das Problem wird durch das sogenannte Bank-Switching gelöst. Der gesamte Speicher ist in Bänke a 64 KByte aufgeteilt.

Durch Beschreiben der Speicherzelle $FF00, des Konfigurations-Registers, kann eine beliebige Bank ein- oder ausgeblendet werden. Die Bänke sind numeriert. Der 128 KByte große RAM-Bereich ist in Bank 0 und Bank 1 unterteilt. Diese RAM-Bänke werden teilweise vom ROM-Bereich (Bank 15) überlagert, in

dem sich die fest eingebauten Programme (Betriebssystem, BASIC-Interpreter, Monitor) befinden.

Überlagert bedeutet, daß der Prozessor an diesen Bereichen das ROM sieht, nicht jedoch das darunterliegende RAM. Mit Hilfe der Speicherzelle $FF00 kann die Speicherkonfiguration beliebig geändert werden. Statt der RAM-Bank 0 kann auf die Bank 1 geschaltet werden, das ROM kann ganz oder teilweise ausgeblendet werden, wodurch der Zugriff auf das darunterliegende RAM möglich wird. Die Änderung der Speicherkonfiguration bereitet jedoch in der Praxis größere Probleme.

Kaum ein Programm kommt ohne Routinen des Betriebssystems (Bank 15) aus. Im Einschaltzustand überlagert das ROM den RAM-Bereich. Das darunterliegende RAM ist überdeckt, der Prozessor sieht es nicht. Sie können nun zwar durch Ändern der mit $FF00 festgelegten Speicherkonfiguration das ROM teilweise ausblenden und auf die komplette 64 KByte große RAM-Bank zugreifen. Leider ist Ihnen anschließend die Benutzung der nun für den Prozessor nicht mehr existenten Betriebssystem-Routinen verwehrt.

Als Lösung bietet Commodore mehrere Routinen an, mit denen Betriebssystem-Routinen aufgerufen werden können, obwohl das ROM ja teilweise ausgeblendet ist. Diese Routinen befinden sich ebenfalls im ROM und zwar beginnen sie am hintersten Ende, ab $FF6E und $FF74. Die Routinen können selbstverständlich nur dann benutzt werden, wenn dieser hintere ROM-Teil nicht ausgeblendet ist (genauer: im Bereich $C000-$FFFF muß das ROM eingeblendet sein)!

An der Adresse $FF74 befindet sich die Routine JSRFAR. Mit einer Unzahl an Parametern müssen Sie dieser Routine vor dem Aufruf unter anderem

- die Adresse der Betriebssytem-Routine, die Sie aufrufen wollen,
- die aktuelle Speicherkonfiguration und
- die gewünschte Speicherkonfiguration mitteilen.

Diese Routine stellt anschließend die gewünschte Speicherkonfiguration her, ruft die jeweilige Betriebssystem-Routine auf, stellt die Originalkonfiguration wieder her und übergibt Ihnen Parameter, die von der eigentlich aufgerufenen Betriebssystem-Routine stammen.

Analog funktioniert die Routine JMPFAR an Adresse $FF74, die die betreffende Betriebssystem-Routine jedoch nicht als Unterprogramm (JSR (ADRESSE)), sondern mit JMP aufruft.

Nachdem ich mich selbst längere Zeit mit wachsender Begeisterung mit den diversen Konfigurationen herumschlug, bin ich der Überzeugung, daß Ihnen dieser Aufwand den Spaß am Erlernen von Assembler gründlich vermiesen würde.

Im folgenden zeige ich Ihnen daher, wohin wir Assembler-Programme legen können, ohne die Standardkonfiguration ändern zu müssen.

*Die Fließkomma-Akkumulatoren ($63-$71)*

Ebenso wie beim C16, Plus/4 kann auch beim C128 der Bereich der sogenannten Fließkomma-Akkumulatoren ($63-$71) in der Zeropage zur vorübergehenden Datenspeicherung verwendet werden.

Aufgrund der kurzen und schnellen Zeropage-Adressierung bietet es sich an, diesen Bereich für Daten zu verwenden, auf die häufig zugegriffen wird. Ein zweiter Anwendungsbereich

sind Zeiger für die indirekt-indizierte Adressierung, die bekanntlich in der Zeropage abgelegt werden müssen!

Beachten Sie jedoch, daß dieser Bereich überschrieben wird, wenn das Assembler-Programm verlassen und nach BASIC zurückgekehrt wird. Sobald der BASIC-Interpreter wieder regiert, verwendet er diesen Bereich selbst für verschiedene Berechnungen.

*Der reservierte Zeropage-Bereich ($FA-$FE)*

Den Bereich $FA-$FE können Sie für Zeiger (indirekt-indizierte Adressierung) oder für den blitzschnellen Zugriff (Zeropage-Adressierung) auf häufig benötigte Daten verwenden.

Alle Daten in diesem Bereich sind vor dem Überschreiben geschützt, da Commdore diese fünf Bytes speziell für Anwender-Software reservierte.

*Der Kassettenpuffer ($0B00-$0BFF)*

Der Bereich $0B00-$0BFF ist unbenutzt, wenn Sie - wie wohl die meisten 128er-Besitzer - nicht mit der Datasette, sondern mit einer Floppy arbeiten.

In diesem Bereich können Sie völlig unbesorgt Daten und kleinere Programme ablegen.

*Die RS-232-Schnittstelle ($0C00-$0DFF)*

Die RS-232-Schnittstelle werden die wenigsten Besitzer eines C128 benutzen. Eine Anwendung dieser Schnittstelle ist der Anschluß und Betrieb eines Modems.

Wenn Sie diese Schnittstelle nicht benötigen, stehen Ihnen mit dem Bereich $0C00-$0DFF volle zwei Pages (512 Byte) zur freien Verfügung.

*Der Bereich $1300-$17FF*

Der Bereich von $1300-$17FF ist unbenutzt und bietet uns ausreichend Platz auch für größere Assembler-Programme. Wie der folgende Abschnitt zeigt, ist dieser Bereich jedoch problematisch, wenn darin gespeicherte Programme zwischen verschiedenen RAM-Bänken umschalten sollen.

*Der Bereich $1800-$1FFF*

Dieser Bereich ist mit Einschränkungen nutzbar. Zum einen gelten alle Einschränkungen, die (siehe folgenden Abschnitt) für den Bereich $1300-$17FF zu machen sind. Außerdem ist dieser Bereich nur bedingt unbenutzt.

$1800-$18FF enthält die Funktionstastenbelegung. Programme in diesem Bereich werden sofort verändert, wenn die Belegung der Funktionstasten mit dem KEY-Befehl geändert wird.

$1C00-$1FFF enthält die sogenannte Bit-Map, wenn die hochauflösende Graphik eingeschaltet ist. Bedingung zur Nutzung dieses Bereichs ist daher der Verzicht auf Graphik.

Alle folgenden Speicherbereiche werden von ROM-Bereichen überlagert und sind daher ohne Änderung der Speicherkonfiguration nicht nutzbar. Sind Ihre Programme sehr umfangreich und wollen Sie daher die folgenden Bereiche benutzen, müssen Sie das überlagernde ROM ausblenden, mit allen genannten Konsequenzen für die Benutzung der ROM-Routinen.

Im folgenden Abschnitt zeige ich akzeptable Lösungen für diese Problematik auf.

## 24.2. Das Bankswitching

Wenn Ihnen die im letzten Abschnitt erläuterten Bereiche ausreichen, können Sie sofort anfangen, im 128er-Modus zu programmieren.

Wollen Sie Assembler-Routinen schreiben, die mit BASIC-Programmen zusammenarbeiten sollen, stellt uns ausgerechnet der größte dieser Abschnitte, der Bereich $1300-$17FF, vor weitere Probleme.

Der BASIC-Interpreter benutzt Bank 0 zur Speicherung des BASIC-Programms und Bank 1 zur Ablage der Variablen. Assembler-Unterprogramme, die zeitkritische BASIC-Teile ersetzen sollen, müssen nun leider sehr häufig auf die BASIC-Variablen in Bank 1 zugreifen (ebenso wie reine Assembler-Programme, die viel Platz für große Datenmengen benötigen).

In dem Moment jedoch, in dem Ihr Programm von Bank 0 auf Bank 1 umschaltet, stürzt es ab. Nach der Umschaltung auf Bank 1 sieht der Prozessor Ihr Programm nicht mehr, da Bank 0 (in dem sich Ihr Programm befindet) ausgeblendet wurde. Es ist für den Prozessor nicht mehr existent.

Einen Ausweg bilden die sogenannten gemeinsamen Speicherbereiche. Ein gemeinsamer Speicherbereich ist ein Bereich, der in jeder Bank den gleichen Inhalt besitzt. Stellen Sie sich zur Veranschaulichung vor, daß der Rechner beim Zugriff auf einen gemeinsamen Bereich die jeweilige Operation in allen (!) RAM-Bänken ausführt. Nach dem Umschalten auf eine andere Bank sieht er daher die gleichen Daten wie zuvor (jedoch nur im als gemeinsam deklarierten Bereich!).

Gemeinsame Speicherbereiche müssen am Anfang oder am Ende des Speichers beginnen. Ausgehend von einem Ende des Speichers können wir selbst in Ein-Kilobyte-Schritten gemeinsame

Bereiche deklarieren, also z.B. den Bereich $0000-$03FF (1 KByte), den Bereich $0000-$07FF (2 KByte), $0000-$0BFF (3 KByte) usw.

Als Standardwert ist beim Einschalten des Rechners der Bereich $0000-$03FF als gemeinsamer Bereich von Bank 0 und Bank 1 deklariert.

Der Inhalt dieses Bereichs ist somit in beiden Bänken identisch. Daher kann ein Programm, das sich in diesem Bereich befindet, jederzeit von Bank 0 auf Bank 1 umschalten und umgekehrt, ohne abzustürzen; für den Prozessor ist es in beiden Bänken vorhanden!

Der Bereich $1300-$17FF, der größte Bereich, den wir zur Verfügung haben, ohne das ROM ausblenden zu müssen, befindet sich jedoch außerhalb dieses als gemeinsam deklarierten Bereichs $0000-$03FF.

Wir deklarieren daher den Bereich $0000-$1FFF als gemeinsamen Bereich. Mitten in diesem Bereich, ab Adresse $1C00 beginnt in Bank 0 der BASIC-Textspeicher (BASIC-Programme). Ebenfalls in diesem Bereich, jedoch in Bank 1, beginnt die sogenannte Variablentabelle des BASIC-Interpreters.

Definieren wir den Bereich $0000-$1FFF als gemeinsam, überschneiden sich BASIC-Programm und Variablentabelle. Sobald unser BASIC-Programm Variablen anlegt, wird nicht nur die Variablentabelle in Bank 1 beschrieben, sondern zugleich das BASIC-Programm in Bank 0 überschrieben.

Daher müssen wir - wie, zeige ich noch - dafür sorgen, daß das BASIC-Programm erst ab $2000 beginnt. Zusätzlich muß auch der Beginn der Variablentabelle verschoben werden, da ab $0800 in Bank 0 der Bildschirmspeicher des C128 beginnt.

Überschneidungen zwischen dem Bildschirmspeicher und der Variablentabelle würden ansonsten dazu führen, daß jede Bildschirmausgabe die Inhalte der Variablentabelle ändert.

Sie sehen bereits, die gemeinsamen Bereiche sind außerordentlich problematisch zu nutzen. Ich stelle Ihnen nun zwei kleine Programme (eine BASIC- und eine Assembler-Routine) vor, die die benötigte Verschiebearbeit für Sie erledigen:

Beginn des BASIC-Textes und der Variablentabelle verschieben

```
100 POKE 46,DEC("20"):POKE 48,DEC("20")
110 POKE DEC("2000"),0
120 CLR:NEW
```

```
Gemeinsamen Speicherbereich deklarieren
A 00B00 AD 00 FF     LDA $FF00
A 00B03 48           PHA
A 00B04 A9 00        LDA #$00
A 00B06 8D 00 FF     STA $FF00
A 00B09 AD 06 D5 LDA $D506
A 00B0C 09 06        ORA #$06
A 00B0E 8D 06 D5     STA $D506
A 00B11 68           PLA
A 00B12 8D 00 FF     STA $FF00
A 00B15 60           RTS
Umschalten auf Bank 0
A 00B16 08           PHP
A 00B17 48           PHA
A 00B18 A9 00        LDA #$00
A 00B1A 8D 00 FF     STA $FF00
A 00B1D 68           PLA
A 00B1E 28           PLP
A 00B1F 60           RTS
Umschalten auf Bank 1
A 00B20 08           PHP
A 00B21 48           PHA
A 00B22 A9 7F        LDA #$7F
A 00B24 8D 00 FF     STA $FF00
A 00B27 68           PLA
A 00B28 28           PLP
A 00B29 60           RTS
```

Speichern Sie diese Programme auf Cassette oder Diskette und gehen Sie von nun an bei der Assembler-Programmierung wie folgt vor:

1. BASIC-Programm laden und starten (=>Beginn des BASIC-Textes und der Variablentabelle werden nach $2000 verschoben).

2. Assembler-Programm laden und mit SYS DEC("0B00") aufrufen (=>Deklarierung eines gemeinsamen Bereichs von $0000-$1FFF).

Der Ablauf des BASIC-Programms: In 46 befindet sich das High-Byte eines Zeigers auf den Beginn des BASIC-Textes. Das Low-Byte (in 45) enthält den Wert $00. Mit POKE 46,DEC("20") wird der Zeiger auf die Adresse $2000 verbogen, den neuen BASIC-Anfang.

Analog verbiegt POKE 48,DEC("20") den Zeiger (in 47/48) auf den Anfang der Variablentabelle ebenfalls auf die Adresse $2000.

Sowohl BASIC-Programme als auch die Variablentabelle beginnen hinter dem nun durch die Assembler-Routine deklarierten gemeinsamen Bereich $0000-$1FFF.

Der Ablauf des Assembler-Programms: Die Routine rettet zuerst die aktuelle Speicherkonfiguration, die sich im Register $FF00 befindet.

Gemeinsame Bereiche werden mit der Speicherzelle $D506 deklariert. Diese Speicherzelle befindet sich im sogenannten I/O-Bereich, der vor einer Änderung eingeblendet werden muß, indem ins Konfigurationsregister $FF00 der Wert $00 geschrieben wird. $D506 kann nun mit dem Wert $06 geort werden, womit der Bereich $0000-$1FFF als gemeinsam deklariert wird. Zum Schluß wird der - auf den Stack gerettete - ursprüngliche Inhalt des Konfigurationsregisters wiederhergestellt.

Sie können nun in alle freien Bereiche (siehe oben) bis $1FFF Assembler-Programme legen, die zwischen den verschiedenen Bänken umschalten, ohne daß diese sich selbst den Ast absägen, da sie in beiden Bänken existent sind.

Auf Bank 1 schalten Sie um, indem Sie das Konfigurationsregister $FF00 mit dem Wert $7F beschreiben. Mit diesem Wert wird Bank 1 (die Variablenbank) ein- und das ROM ausgeblendet, so daß Sie auf die vollen 64 KByte von Bank 1 zugreifen können. Auf Bank 0 schalten Sie zurück, indem Sie $FF00 mit $00 beschreiben.

Beide Umschaltroutinen sind in der Assembler-Routine bereits in sehr komfortabler Form enthalten. Komfortabel, da beide Routinen weder Registerinhalte noch das Status-Register verändern (bzw. auf den Stack bringen und anschließend zurückholen).

Beide Routinen können Sie daher von jeder beliebigen Stelle Ihres Programms aus aufrufen, ohne Veränderungen irgendwelcher Register oder Flags befürchten zu müssen.

Mit JSR $0B20 schalten Sie auf Bank 1 (nur RAM) und mit JSR $0B16 auf Bank 0 zurück (Standardkonfiguration RAM + ROM).

Zusammenfassung

1. Wenn Sie reine Assembler-Programme schreiben, können Sie ohne Änderung der Speicherkonfiguration alle genannten freien Speicherbereiche nutzen.

2. Benötigt Ihr Assembler-Programm zusätzlichen Speicherplatz für große Datenmengen, müssen Sie während des Zugriffs auf die Daten das ROM ausblenden und können entweder Bank 0 für Ihre Daten verwenden oder aber auf Bank 1 umschalten.

3. Assembler-Routinen, die mit BASIC-Programmen zusammenarbeiten sollen, müssen (!) häufig auf Bank 1 umschalten, da der BASIC-Interpreter Variablen in dieser Bank ablegt.

4. Gehen Sie in allen Programmen, die auf Bank 1 umschalten müssen, wie folgt vor:

- Laden und Starten Sie das vorgestellte BASIC-Programm.
- Laden Sie das ebenfalls vorgestellte Assembler-Programm und rufen Sie es mit SYS DEC("0B00") auf.
- Mit JSR $0B20 rufen Sie die integrierte Routine zum Umschalten auf Bank 1 (nur RAM auf).
- Mit JSR $0B16 rufen Sie die Routine zum Zurückschalten auf Bank 0 (RAM + ROM) auf.

Mit dieser Vorgehensweise und den Umschaltroutinen können Sie jederzeit wie gewohnt die Betriebssystem-Routinen benutzen und - dank der Umschaltroutinen - die vollen 64 KByte von Bank 1.

Für Ihre Programme stehen Ihnen alle erwähnten freien Bereiche ohne Einschränkungen zur Verfügung.

## 24.3. Bildschirmausgaben

Auf den letzten Seiten sahen sich C128-Besitzer mit sehr speziellen Problemen Ihres Rechners konfrontiert. Die verschiedensten Speicherkonfigurationen in aller Ausführlichkeit zu besprechen, überschreitet zweifellos den Rahmen eines Assembler-Lehrbuchs. Für detaillierte Informationen bin ich daher gezwungen, Sie auf spezielle C128-Lektüre zu verweisen.

Bevor Sie sich auf die folgenden Kapiteln stürzen, will ich Sie zuvor noch über eine Eigenheit Ihres Bildschirmspeichers aufklären. Wie Sie wissen, bietet der C128 zwei verschiedene Ausgabemodi, den 40- und den 80-Zeichen-Modus.

Im 40-Zeichen-Modus befindet sich der Bildschirmspeicher des C128 ebenso wie beim C64 im Bereich $0400-$07FF. Auf die einzelnen Speicherzellen kann wie üblich lesend oder schreibend zugegriffen werden.

Der 80-Zeichen-Modus erlaubt leider keinen direkten Zugriff auf den Bildschirmspeicher. Die Ursache dieses unterschiedlichen Verhaltens liegt in dem verwendeten Video-Controller. Für den 40-Zeichen-Modus ist der VIC, für den 80-Zeichen-Modus der VDC zuständig.

Letzterer gestattet leider nur den indirekten Zugriff auf den Bildschirmspeicher über bestimmte Register (Speicherzellen des Videocontroller-Chips). Das Ganze funktioniert mal wieder reichlich umständlich.

Der VDC besitzt verschiedene Register, die wir ansprechen müssen. Die Register müssen jedoch wiederum indirekt angesprochen werden. Zuerst muß $D600, das Kontroll-Register, mit der Nummer des betreffenden Registers beschrieben werden.

Anschließend müssen wir in einer Schleife Bit 7 des Registers $D600 testen und abwarten, bis es gelöscht ist.

Nun ist das Daten-Register beschreibbereit und in dieses wird der Wert geschrieben, den wir dem eigentlich anzusprechenden Register übermitteln wollen. Die Nummern der uns interessierenden Register bei der Zeichenausgabe:

$13: Adresse Low der Bildschirmadresse, die wir ansprechen wollen.
$12: Adresse High der Bildschirmadresse.
$1F: Nachdem die Adresse übergeben wurde, befindet sich das Zeichen an der betreffenden Bildschirmposition in diesem Register und kann gelesen werden. Beschreiben des Registers führt zur Ausgabe des betreffenden Zeichens.

Der Bildschirmspeicher besitzt im 80-Zeichen-Modus die Adressen $0000-$07FF. Um ein Zeichen an Adresse $0A (Spalte 10/Zeile 0) auszugeben, gehen Sie wie folgt vor:

- $13 in $D600 schreiben und warten, bis Bit 7 von $D600 gelöscht ist.
- Low-Byte von $000A in $D601 schreiben.
- $12 in $D600 schreiben und warten, bis Bit 7 von $D600 gelöscht ist.
- High-Byte von $000A in $D601 schreiben.
- $1F in $D600 schreiben und warten, bis Bit 7 von $D600 gelöscht ist.
- $D601 auslesen oder mit dem auszugebenden Zeichen beschreiben.

Ich gebe Ihnen jedoch einen Tip: ersparen Sie sich den Aufwand. Üblicherweise greift man aus Geschwindigkeitsgründen direkt auf den Bildschirmspeicher zu. Der direkte Zugriff ist weitaus schneller als die Ausgabe mit BSOUT.

Dies gilt jedoch nicht für den 80-Zeichen-Modus des C128. Aufgrund der Arbeitsweise des Videocontrollers ist die Bildschirmausgabe auch mit dem erläuterten indirekten direkten Zugriff nur unwesentlich schneller als mit BSOUT.

Vergessen Sie daher am besten alles, was ich zu diesem Thema sagte und verwenden Sie BSOUT, wenn Sie sich Frustrationen - die ich bereits hinter mir habe - ersparen wollen.

## 25. Programmieren mit Monitor und Assembler

Die Überschrift dieses Kapitels "Programmieren mit Monitor und Assembler" deutet bereits eine Begriffsverwirrung an. Wir programmieren doch bereits in - der Sprache - Assembler, wie sollen wir nun mit (!) einem Assembler programmieren?

Unter einem Assembler versteht man ein Hilfsprogramm namens Assembler, das die Programmierung in der Sprache (!) Assembler erheblich erleichtert.

Wie in diesem Buch deutlich wurde, gibt es verschiedene Ebenen der Assembler-Programmierung. Auf der niedrigsten Ebene programmieren Sie, wenn Sie der Reihe nach alle Befehlscodes und Argumente Ihres Programms von BASIC aus in den Speicher poken.

```
POKE 49152,169:REM BEFEHLSCODE FUER 'LDA #(WERT)'
POKE 49153,10:REM ARGUMENT $0A
POKE 49154,00:REM BEFEHLSCODE FUER 'BRK'
```

Diese drei POKE-Befehle schreiben die Befehle

```
LDA #$0A
BRK
```

ab 49152 ($C000) in den Rechnerspeicher. Ein Monitor erlaubt eine weit komfortablere Programmerstellung, da er Assembler-Befehle und Argumente selbständig in die zugehörigen Zahlencodes umwandelt und ablegt.

Die Programmierung mit einem Monitor befindet sich eine Ebene über dem Poken eines Assembler-Programms. Zusätzlich stellt uns der Monitor sehr komfortable Hilfen zum Listen, Speichern und Laden eines Programms zur Verfügung.

Diese Hilfen bietet ein Assembler nicht. Dafür ist ein solches Programm ein außerordentlich effizientes Werkzeug zur Programmeingabe. Einer der wesentlichsten Vorteile eines Assemblers besteht in der Verwendung der Label.

Label können Sie mit Variablen in einem BASIC-Programm vergleichen. Dem Label wird am Programmanfang ein Wert oder eine Adresse zugewiesen und im weiteren Prorgrammtext nur noch das Label verwendet. Das erstellte Programm befindet sich

jedoch noch nicht in lauffähiger Form im Speicher. Unter Verwendung von Label schreiben wir zuerst einen sogenannten Source-Text, der z.B. so aussehen kann:

```
110 ;*** REM LABELDEKLARATION ***
110 .EQ BSOUT=$FFD2
120 .EQ GETIN=$FFE4
130 .EQ PLOT =$FFF0
140 ;
150 .BA $C000
160 ;
170 ;*** PROGRAMM ***
180 LDX #$02
190 LDY #$0A
200 CLC
210 JSR PLOT
220 LDA #$41
230 JSR BSOUT
240 BRK
250 ;
260 .EN
```

Der Source-Text ist ähnlich aufgebaut wie ein BASIC-Programm. Vor dem eigentlichen Befehl geben wir eine Zeilennummer ein, die die Reihenfolge der Befehle festlegt.

Am Anfang des Textes deklarieren wir die benötigten Label, d.h., wir geben jeder häufig verwendeten Adresse (oder jeden Wert) einen Namen und verwenden im folgenden Programm dieses Label als Stellvertreter für die Adresse oder den Wert.

Im Source-Text befindet sich immer eine spezielle Anweisung, die die Startadresse des Programms kennzeichnet (hier: .BA $C000 => Startadresse $C000) und ein weiterer Befehl, der das Programmende markiert (hier: .EN).

Der Programmtext wird auf eine - von Assembler zu Assembler variierende - Anweisung hin assembliert, d.h., in ein lauffähiges Assembler-Programm übersetzt.

Der Assembler (das Hilfsprogramm) geht alle Befehlszeilen durch und legt ab $C000 die entsprechenden Befehlscodes und Argumente im Speicher ab. Label werden durch jene Werte ersetzt, die Ihnen am Anfang des Textes zugewiesen wurden. Stößt der Assembler auf das Label BSOUT, wird er die diesem Label zugewiesene Adresse $FFD2 im Speicher ablegen.

Der Unterschied zwischen einem Monitor und einem Assembler besteht darin, daß ersterer Befehlscodes und Argumente sofort (!) im Speicher ablegt, während letzterer erst auf Kommando einen Programmtext bearbeitet und in ein lauffähiges Programm umwandelt.

Sie können die Arbeitsweise eines Assemblers in etwa mit einem BASIC-Compiler vergleichen, der ja ebenfalls aus einem BASIC-Text ein echtes Assembler-Programm erzeugt.

Mit einem Assembler befinden wir uns somit eine Ebene über der Programmeingabe mit einem Monitor. Mit Befehlscodes haben wir überhaupt nichts mehr zu tun und dank der einmaligen Deklarierung von Label auch sehr wenig mit Adressen. Statt $FFD2 verwenden wir ausschließlich das Label BSOUT, statt $FFF0 das Label PLOT usw..

Mit einem Assembler wird die Eingabe eines Maschinenprogramms so komfortabel wie in einer höheren Programmiersprache, vor allem, da auch für Sprungziele Label eingesetzt werden können.

```
300      LDX #$FF
310 LOOP STA $0400,X
320      DEX
330      BNE LOOP
```

Bei der Assemblierung berechnet der Assembler die Sprungweite des Branches BNE und legt im Speicher den Befehlscode von BNE ($D0) zusammen mit dem korrekten Argument ($FA) ab.

Im Gegensatz zur Programmeingabe mit einem Monitor können wir unsere Programme nachträglich problemlos verändern und z.B. fehlende Befehle in die Schleife einfügen. Wir müssen keine Sprungadressen ändern, diese Aufgabe übernimmt der Assembler, den wir anschließend anweisen, den Programmtext erneut zu assemblieren.

Wahrscheinlich werden Sie nun mehr oder weniger verärgert über mich sein. Immerhin hätte ich Sie bereits am Anfang dieses Buches über diese außerordentlich komfortable Möglichkeit zur Erstellung von Assembler-Programmen informieren können und Ihnen wäre die vergleichsweise primitive Programmeingabe mit dem Monitor erspart geblieben.

Bedenken Sie jedoch, daß die Programmeingabe mit einem Monitor sehr viel direkter ist als das Erstellen eines Textes, den ein anonymes Programm für Sie in Maschinensprache umwandelt.

Ich selbst begann mit einem Assembler zu programmieren und habe dies im nachhinein des öfteren bedauert. Wenn Sie ausschließlich mit Label arbeiten, werden Sie niemals durchschauen, was unter einem relativen Sprung zu verstehen ist, im Gegensatz zur Monitoreingabe BNE $C000, die Ihnen sofort das relative Argument anzeigt.

Sie werden niemals so unmittelbar wie mit dem Monitor erfahren, was passiert, wenn ein Register oder eine Speicherzelle den Wert $FF enthält und nun inkrementiert wird.

```
A C000 LDX #$FF
A C002 INX
A C003 BRK
```

Sie werden theoretisch zwar wissen, wie Adressen im Speicher abgelegt werden (Low-Byte/High-Byte), bei der Eingabe mit dem Monitor sehen (!) Sie das Adreßformat jedoch sofort.

```
A C000 AD 00 04  LDA $0400
```

Ich bin daher der Meinung, daß ein Assembler erst dann benutzt werden sollte, wenn bereits solide Assembler-Kenntnisse und viel Übung vorhanden sind.

An diesem Punkt sind Sie nun angelangt. Wenn Sie wollen, können Sie sich ein solches Programm besorgen und in den folgenden Kapiteln sofort einsetzen. Hinweise zur Bedienung des Assemblers kann ich Ihnen nicht geben, da die speziellen Befehle - z.B. zur Label-Deklaration - leider nicht so standardisiert sind wie bei Monitor-Programmen.

Auch in den folgenden Kapiteln werde ich jedoch wie bisher mit dem Monitor arbeiten und es schadet bestimmt nicht, wenn Sie sich auch weiterhin mit dieser primitiven Form der Programmeingabe begnügen.

Der Einsatz eines Assemblers ist jedoch fast zwingend notwendig, wenn Sie größere Programme erstellen. Kleine Programme wie die in diesem Buch vorgestellten können ebensogut mit einem Monitor eingegeben werden.

In der Praxis wird man meist mit beiden Hilfsmitteln arbeiten. Zur Programmerstellung verwendet man den Assembler und zur Überprüfung des Programmablaufs den Monitor.

Eine einfache Möglichkeit, den Monitor zur Programmüberprüfung einzusetzen, bietet der BRK-Befehl. Sie schreiben ein Programm, assemblieren es und stellen nach dem Aufruf fest, daß sich noch Fehler im Programm befinden.

Eine übliche Methode zur Fehlersuche besteht in BASIC-Programmen darin, an jener Stelle, an der man einen Fehler vermutet, ein STOP einzufügen und nach dem Programmabbruch die Inhalte diverser Variablen mit PRINT zu überprüfen.

An jener Adresse, an der Sie in Ihrem Assembler-Programm den Fehler vermuten, geben Sie analog den Befehl BRK ein und schauen sich nach dem Programmabbruch die Registeranzeige

oder aber, mit dem Memory-Befehl (M), die Inhalte wichtiger Speicherzellen an.

Läuft das Programm bis zu diesem Punkt korrekt, ersetzen Sie den BRK-Befehl durch den ursprünglichen Assembler-Befehl und setzen an einem folgenden Programmpunkt einen BRK.

Nun stellt sich jedoch oft das Problem, daß Sie nicht mehr wissen, welcher Befehl durch den BRK-Befehl überschrieben wurde. In diesem Fall bleibt Ihnen nur eine Möglichkeit, die komplette Neuassemblierung des gesamten Programms.

Eine elegantere Methode besteht darin, im Source-Text an allen Stellen, an denen später eventuell BRK-Befehle einzufügen sind, den Befehl NOP einzugeben:

```
200       LDX #$FF
210 LOOP  STA $0400,X
220       DEX
230       BPL LOOP
240       NOP
250       LDA $D020
260       LDX $D021
270       NOP
```

Der Befehl NOP besitzt den Befehlscode $EA und benötigt kein weiteres Argument. Dieser Befehl hat keinerlei Auswirkungen, er beeinflußt weder irgendwelche Register noch Speicherzellen.

Daher kann dieser Befehl bei der späteren Fehlersuche mit dem Monitor problemlos durch BRK überschrieben und hinterher wieder durch NOP ersetzt werden.

# Teil 4: Zwei Welten kommen sich näher: BASIC und Assembler

Mehrmals in diesem Buch versprach ich Ihnen, auf ein Thema einzugehen, das leider in kaum einem Assembler-Lehrbuch beschrieben wird, auf die Zusammenarbeit von Assembler-Routinen und BASIC-Programmen.

In folgenden Teil wird Ihnen das nötige Handwerkszeug vermittelt, um z.B. Eingabe- oder Sortierroutinen in Assembler zu schreiben, die von einem BASIC-Programm genutzt werden.

Die in diesem dritten Teil entwickelten Programme sind deutlich komplexer als die vorangegangenen und werden deshalb in verschiedenen Teilschritten entwickelt und erläutert. Am Ende eines jeden Abschnitts wird das Listing des kompletten Programms vorgestellt.

Vor der Entwicklung diverser Programme ist jedoch ein wenig Theorie unumgänglich, da Sie unbedingt darüber Bescheid wissen müssen, wie der BASIC-Interpreter einen BASIC-Text liest und Variablen organisiert.

## 26. Wie BASIC-Programme abgelegt werden

BASIC-Programme legt der Interpreter in einer sehr speziellen Form im BASIC-RAM ab. Beim C64 beginnt der BASIC-Speicher ab $0400. An dieser Adresse befindet sich das Byte $00, an dem das folgende BASIC-Programm erkannt wird.

Den Beginn jeder Programmzeile bilden zwei Link-Bytes, die die Adresse (in der Form Low-Byte/High-Byte) der Link-Bytes der nächsten Programmzeile enthalten.

Über diese Link-Bytes kann sich der Interpreter sehr schnell durch ein BASIC-Programm hangeln, ohne jeweils die komplette Programmzeile lesen zu müssen.

Den Link-Bytes folgt die Zeilennummer (ebenfalls zwei Bytes), der wiederum der eigentliche Programmtext folgt. Der Programmtext wird keineswegs Zeichen für Zeichen im Speicher abgelegt, sondern platzsparender. Jedem BASIC-Befehle wird ein Ein-Byte-Wert zugeordnet, das sogenannte Token, der den Befehl eindeutig identifiziert.

Statt PRINT wird daher das zugehörige Token abgelegt. Abgesehen von den BASIC-Befehlen wird der restliche Programmtext in ASCII-Form Zeichen für Zeichen abgelegt. Das Ende einer Programmzeile wird immer am Code $00 erkannt.

Auf diese Weise wird Zeile für Zeile im Speicher abgelegt. Das Programmende markiert eine zusätzliche null ($00 $00). Zum Beispiel wird beim C64 das Programm

```
100 PRINT "A"
110 PRINT "B"
```

wie folgt abgelegt:

```
0800   00 0B 08 64 00 99 20 22 =>  Link-B./Zeilennr./PRINT "
0808   41 22 00 15 08 6E 00 99 =>  A"/00/Link-B.Zeilennr./PRINT
0810   20 22 42 22 00 00       =>  "B"/00/00
```

Für Sie ist die Art und Weise, wie der Interpreter ein BASIC-Programm ablegt, interessant, wenn Sie z.B. Hilfsprogramme wie einen REM-Killer schreiben wollen, ein Programm, das automatisch alle REM-Zeilen aus einem BASIC-Programm entfernt.

## 27. Variablenbehandlung durch den Interpreter

Die Art und Weise, wie der Interpreter ein BASIC-Programm im Speicher ablegt, wird uns nicht weiter beschäftigen.

Außerordentlich wichtig für uns ist jedoch die Art und Weise, wie der Interpreter Variablen behandelt. Ohne dieses Wissen können wir niemals eine Sortierroutine schreiben, die z.B. ein Stringarray bearbeitet (oder wissen Sie bereits, wo das Assembler-Programm die Strings zu suchen hat?).

Wir müssen drei Variablentypen unterscheiden: Integer-, Fließkomma- und Stringvariablen. Bei allen drei Typen sind wiederum einfache und Feld- oder Array-Variablen zu unterscheiden.

Die folgenden Erläuterungen gelten uneingeschränkt für den C64. Kleine Unterschiede beim C16, Plus/4 und etwas größere beim C128 werden im Anschluß an dieses Kapitel beschrieben.

### Einfache Variablentypen

Unmittelbar hinter einem BASIC-Programm befindet sich die sogenannte Variablentabelle. Diese Tabelle enthält alle einfachen und dimensionierten numerischen Variablen (Integer/Fließkomma).

Jeder Eintrag in dieser Tabelle ist bei nicht dimensionierten Variablen genau sieben Byte lang. Die ersten beiden Byte (Byte 1 und 2) enthalten den jeweiligen Variablennamen und kennzeichnen zugleich den Variablentyp.

*Integervariablen*: Bit 7 der beiden Namensbytes sind gesetzt.
*Fließkommavariablen*: Bit 7 beider Namensbytes ist gelöscht.
*Stringvariablen*: Bit 7 ist nur im zweiten Namensbyte gesetzt.

Die restlichen fünf Bytes (Byte 3-7) besitzen folgende Bedeutung:

*1. Integervariablen*

*Byte 3 und 4*: Zahlenwert in der Form High-Byte/Low-Byte.
*Byte 5-7*: ungenutzt.

Beachten Sie bitte die ungewohnte Form, in der eine Integerzahl gespeichert wird. Zuerst kommt das High-Byte, anschließend das Low-Byte, nicht umgekehrt wie beim Adreßformat.

*2. Fließkommavariablen*

*Byte 3-7*: Zahlenwert im sogenannten Fließkomma-Format, in dem eine Zahl durch fünf Byte dargestellt wird.

*3. Stringvariablen*

*Byte 3*: Stringlänge (0-255)
*Byte 4-5*: Zeiger auf die Adresse des Strings
*Byte 6-7*: ungenutzt

Die Strings selbst sind nicht (!) in der Variablentabelle enthalten, sondern befinden sich am Ende (!) des BASIC-RAM's. Der erste String wird am Ende des BASIC-Speichers angelegt, der nächste String unmittelbar vor diesem usw..

Man spricht auch vom String-Stack, der nach unten - in Richtung niedrigerer Adressen - wächst. Die Variablentabelle enthält ein Byte (Byte 3), das Auskunft über die Länge des Strings gibt, und zwei weitere Byte (4 und 5), die einen Zeiger auf den String darstellen (im Adreßformat Low-Byte/High-Byte).

Daß die Strings nicht ebenso wie numerische Variablen in der Variablentabelle enthalten sind, sondern nur die sogenannten Stringdescriptoren, hat einen einleuchtenden Grund.

Wie erläutert, beginnt die Variablentabelle unmittelbar hinter dem BASIC-Programm. Angenommen, eine Programmzeile wird gelöscht. Dann muß die gesamte Tabelle nach unten verschoben werden, hinter das neue Ende des BASIC-Programms, eine langwierige Aufgabe, wenn die Tabelle eine Vielzahl großer Zeichenketten enthält.

Daher befinden sich die Strings am entgegengesetzten Ende des Speichers, um bei Änderungen der Variablentabelle oder des BASIC-Programms Zeitprobleme durch das Verschieben umfangreicher Zeichenketten zu vermeiden.

**Arrayvariablen**

Arrayvariablen werden in einer komprimierten Form gespeichert. Da die Namen von Arrayvariablen (bis auf den Index) immer identisch sind, wäre die Speicherung von jeweils zwei Namensbyte für jede Arrayvariable eine enorme Platzverschwendung.

Es bietet sich an, nach einem DIM-Befehl sofort Platz für alle Variablen in der Tabelle zu reservieren und nur einmal - am Anfang des reservierten Bereichs - den Namen des Arrays zu speichern.

Genau so geht der BASIC-Interpreter tatsächlich vor. Der Name wird nur einmal am Arrayanfang vermerkt, wobei für die beiden Namensbyte die gleichen Konventionen wie bei einfachen Variablen gelten. Der Arraykopf enthält außer dem Arraynamen noch zwei Byte, die den Speicherplatz angeben, den das Array belegt, ein Byte mit der jeweiligen Dimensionierung, und zwei Byte, die die Anzahl der Arrayelemente enthalten. Erst nach dieser Arraybeschreibung kommen die eigentlichen Arrayvariablen.

*1. Integervariablen*

Für dimensionierte Integervariablen werden nur zwei Byte benötigt, das High-Byte und das Low-Byte der betreffenden Zahl. Diesen zwei Byte folgt unmittelbar die nächste Integervariable des Arrays.

*2. Fließkommavariablen*

Den ersten fünf Byte für die erste Fließkommavariable folgt sofort die zweite Variable usw.

*3. Stringvariablen*

Dimensionierte Stringvariablen benötigen pro Eintrag in der Variablentabelle drei Byte, ein Byte für die Stringlänge und zwei Byte für den Zeiger auf den String.

Bisher gelten alle Ausführungen gleichermaßen für den C64, den C16, Plus/4 und den C128, mit einer kleinen Einschränkung für letzteren. Beim C128 folgt die Variablentabelle nicht unmittelbar im Anschluß auf das BASIC-Programm (Bank 0), sondern beginnt immer ab Adresse $0400 in Bank 1 (Variablenbank).

# Teil 4: Anhang

## 28. Der Befehlssatz

Der Befehlssatz Ihres Rechners ist auf den folgenden Seiten komplett dargestellt. Sie finden in der Tabelle alle möglichen Adressierungsarten der einzlnen Befehle, die zugehörigen Befehlscodes, die Befehlslänge (inclusive Argument) und die Anzahl der Taktzyklen.

In Taktzyklen wird die Ausführungszeit eines Befehls gemessen. Die angegebenen Werte sind als Anhaltspunkte zu verstehen, da in bestimmten Fällen ein oder zwei weitere Taktzyklen benötigt werden, vor allem bei sogenannten Seitenüberschreitungen.

Ein Beispiel: Die Branch-Befehle benötigen mindestens zwei Taktzyklen. Ist die jeweilige Bedingung erfüllt und wird verzweigt, kommt ein weiterer Taktzyklus hinzu. Wird eine Seitengrenze überschritten (Seitenkonzept der Speicherorganisation) werden weitere zwei Taktzyklen benötigt.

Wie lange ein Taktzyklus dauert, ist von Rechner zu Rechner unterschiedlich und hängt von dessen Taktfrequenz ab, die bei Commodore-Rechnern und deren Prozessoren meist ein Megahertz beträgt. Bei einer Taktfrequenz von einem Megahertz finden in einer Sekunde eine Million Taktzyklen statt.

Beispiele zu den Adressierungsarten

```
Implizit:                TAX
Akkumulator:             ASL
Absolut:                 LDA $0400
Zeropage:                LDA $FF
Unmittelbar:             LDA #$0A
X-indiziert:             LDA $0400,X
Y-indiziert:             LDA $0400,Y
indirekt-X-indiziert:    LDA ($FB,X)
indirekt-Y-indiziert:    LDA ($FB),Y
Zeropage-X-indiziert:    LDA $FF,X
Zeropage-Y-indiziert:    LDX $FF,Y
Relativ:                 BNE $C01C
Indirekt:                JMP ($FFFE)
```

## 28.1. Transfer-Befehle

*Befehl*: LDA

*Funktion*:
Lädt den Akkumulator mit dem angegebenen Wert oder dem Inhalt der adressierten Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | AD | 3 | 4 |
| Zeropage: | A5 | 2 | 3 |
| Unmittelbar: | A9 | 2 | 2 |
| X-indiziert: | BD | 3 | 4 |
| Y-indiziert: | B9 | 3 | 4 |
| indirekt-X-indiziert: | A1 | 2 | 6 |
| indirekt-Y-indiziert: | B1 | 2 | 5 |
| Zeropage-X-indiziert: | B5 | 2 | 4 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: LDX

*Funktion*:
Lädt das X-Register mit dem angegebenen Wert oder dem Inhalt der adressierten Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | AE | 3 | 4 |
| Zeropage: | A6 | 2 | 3 |
| Unmittelbar: | A2 | 2 | 2 |
| X-indiziert: | | | |

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Y-indiziert: | BE | 3 | 4 |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | B6 | 2 | 4 |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: LDY

*Funktion*:
Lädt das X-Register mit dem angegebenen Wert oder dem Inhalt der adressierten Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | AC | 3 | 4 |
| Zeropage: | A4 | 2 | 3 |
| Unmittelbar: | A0 | 2 | 2 |
| X-indiziert: | BC | 3 | 4 |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | B4 | 4 | 2 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: STA

*Funktion*:
Überträgt den Inhalt des Akkumulators in die adressierte Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | 8D | 3 | 4 |
| Zeropage: | 85 | 2 | 3 |
| Unmittelbar: | | | |
| X-indiziert: | 9D | 3 | 5 |
| Y-indiziert: | 99 | 3 | 5 |
| indirekt-X-indiziert: | 81 | 2 | 6 |
| indirekt-Y-indiziert: | 91 | 2 | 6 |
| Zeropage-X-indiziert: | 95 | 2 | 4 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: STX

*Funktion*:
Überträgt den Inhalt des X-Registers in die adressierte Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | 8E | 3 | 4 |
| Zeropage: | 86 | 2 | 3 |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | 96 | 2 | 4 |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: STY

*Funktion*:
Überträgt den Inhalt des Y-Registers in die adressierte Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | BC | 3 | 4 |
| Zeropage: | 84 | 2 | 3 |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | 94 | 4 | 2 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: TAX

*Funktion*:
Überträgt den Inhalt des Akkumulators in das X-Register.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | AA | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: TAY

*Funktion*:
Überträgt den Inhalt des Akkumulators in das Y-Register.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | A8 | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: TXA

*Funktion*:
Überträgt den Inhalt des X-Registers in den Akkumulator.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | BA | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |

```
indirekt-Y-indiziert:
Zeropage-X-indiziert:
Zeropage-Y-indiziert:
Relativ:
Indirekt:

Beeinflußte Flags N-Flag  V-Flag  B-Flag  D-Flag  I-Flag  Z-Flag  C-Flag
                    X                                       X
```

*Befehl*: TYA

*Funktion*:
Überträgt den Inhalt des Y-Registers in den Akkumulator.

```
Adressierungsarten      Code   Bytes   Taktzyklen

Implizit:               98      1        2
Absolut:
Zeropage:
Unmittelbar:
X-indiziert:
Y-indiziert:
indirekt-X-indiziert:
indirekt-Y-indiziert:
Zeropage-X-indiziert:
Zeropage-Y-indiziert:
Relativ:
Indirekt:

Beeinflußte Flags N-Flag  V-Flag  B-Flag  D-Flag  I-Flag  Z-Flag  C-Flag
                    X                                       X
```

*Befehl*: TSX

*Funktion*:
Überträgt den Inhalt des Stapelzeigers in das X-Register.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | BA | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: TXS

*Funktion*:
Überträgt den Inhalt des X-Registers in den Stapelzeiger.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 9A | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: PHA

*Funktion*:
Bringt den Inhalt des Akkumulators auf die Stapelspitze.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 48 | 1 | 3 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: PLA

*Funktion*:
Lädt den Akkumulator mit dem obersten Stapel-Element.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 68 | 1 | 4 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: PHP

*Funktion*:
Bringt den Inhalt des Status-Registers auf die Stapelspitze.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 08 | 1 | 3 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: PLP

*Funktion*:
Überträgt das oberste Stapel-Element in das Status-Register.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 28 | 1 | 4 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | X | X | X | X | X | X |

## 28.2. Inkrementier-/Dekrementier-Befehle

*Befehl*: DEC

*Funktion*:
Dekrementiert den Inhalt der angegebenen Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | CE | 3 | 6 |
| Zeropage: | C6 | 2 | 5 |
| Unmittelbar: | | | |
| X-indiziert: | DE | 3 | 7 |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | D6 | 2 | 6 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: DEX

*Funktion*:
Dekrementiert den Inhalt des X-Registers.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | CA | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |

```
Zeropage-X-indiziert:
Zeropage-Y-indiziert:
Relativ:
Indirekt:

Beeinflußte Flags N-Flag  V-Flag  B-Flag  D-Flag  I-Flag  Z-Flag  C-Flag
                    X                                       X
```

*Befehl*: DEY

*Funktion*:
Dekrementiert den Inhalt des Y-Registers.

```
Adressierungsarten      Code   Bytes   Taktzyklen

Implizit:               88       1         2
Absolut:
Zeropage:
Unmittelbar:
X-indiziert:
Y-indiziert:
indirekt-X-indiziert:
indirekt-Y-indiziert:
Zeropage-X-indiziert:
Zeropage-Y-indiziert:
Relativ:
Indirekt:

Beeinflußte Flags N-Flag  V-Flag  B-Flag  D-Flag  I-Flag  Z-Flag  C-Flag
                    X                                       X
```

*Befehl*: INC

*Funktion*:
Inkrementiert den Inhalt der adressierten Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | EE | 3 | 6 |
| Zeropage: | E6 | 2 | 5 |
| Unmittelbar: | | | |
| X-indiziert: | FE | 3 | 7 |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | F6 | 2 | 6 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: INX

*Funktion*:
Inkrementiert den Inhalt des X-Registers.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | E8 | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: INY

*Funktion*:
Inkrementiert den Inhalt des Y-Registers.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | C8 | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

## 28.3. Vergleichsbefehle

*Befehl*: CMP

*Funktion*:
Vergleicht den Inhalt des Akkumulators mit dem angegebenen Wert oder dem Inhalt der adressierten Speicherzelle. Der Wert wird vom Akkumulator subtrahiert und die Flags entsprechend dem Ergebnis gesetzt. Anschließend wird der ursprüngliche Akkumulator-Inhalt wiederhergestellt.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | CD | 3 | 4 |
| Zeropage: | C5 | 2 | 3 |
| Unmittelbar: | C9 | 2 | 2 |
| X-indiziert: | DD | 3 | 4 |
| Y-indiziert: | D9 | 3 | 4 |
| indirekt-X-indiziert: | C1 | 2 | 6 |
| indirekt-Y-indiziert: | D1 | 2 | 5 |
| Zeropage-X-indiziert: | D5 | 2 | 4 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | X |

*Befehl*: CPX

*Funktion*:
Vergleicht den Inhalt des X-Registers mit dem angegebenen Wert oder dem Inhalt der adressierten Speicherzelle. Der Wert wird vom X-Register subtrahiert und die Flags entsprechend dem Ergebnis gesetzt. Anschließend wird der ursprüngliche Register-Inhalt wiederhergestellt.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | EC | 3 | 4 |
| Zeropage: | E4 | 2 | 3 |
| Unmittelbar: | E0 | 2 | 2 |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | X |

Befehl: CPY

Funktion: Vergleicht den Inhalt des Y-Registers mit dem angegebenen Wert oder dem Inhalt der adressierten Speicherzelle. Der Wert wird vom Y-Register subtrahiert und die Flags entsprechend dem Ergebnis gesetzt. Anschließend wird der ursprüngliche Register-Inhalt wiederhergestellt.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | CC | 3 | 4 |
| Zeropage: | C4 | 2 | 3 |
| Unmittelbar: | C0 | 2 | 2 |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | X |

## 28.4. Branch- und absolute Sprungbefehle

*Befehl*: BEQ

*Funktion*:
Verzweigt zur - relativ - angegebenen Adresse, wenn das Zero-Flag gelöscht ist (Ergebnis der letzten Operation gleich null).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | F0 | 2 | 2 |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: BNE

*Funktion*:
Verzweigt zur - relativ - angegebenen Adresse, wenn das Zero-Flag gesetzt ist (Ergebnis der letzten Operation ungleich null).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | D0 | 2 | 2 |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: BPL

*Funktion*:
Verzweigt zur - relativ - angegebenen Adresse, wenn das Negativ-Flag gelöscht ist (Ergebnis der letzten Operation ist positiv (<128)).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | 10 | 2 | 2 |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: BMI

*Funktion*:
Verzweigt zur - relativ - angegebenen Adresse, wenn das Negativ-Flag gesetzt ist (Ergebnis der letzten Operation ist negativ (>=128)).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | 30 | 2 | 2 |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: BCC

*Funktion*:
Verzweigt zur - relativ - angegebenen Adresse, wenn das Carry-Flag gelöscht ist (Ergebnis der letzten Operation ergab keinen Übertrag).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | 90 | 2 | 2 |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: BCS

*Funktion*:
Verzweigt zur - relativ - angegebenen Adresse, wenn das Carry-Flag gesetzt ist (Ergebnis der letzten Operation ergab einen Übertrag).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | B0 | 2 | 2 |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: BVC

*Funktion*:
Verzweigt zur - relativ - angegebenen Adresse, wenn das V-Flag gelöscht ist (Ergebnis der letzten Operation ergab keinen Überlauf von Bit 6 in Bit 7).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | 50 | 2 | 2 |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: BVS

*Funktion*:
Verzweigt zur - relativ - angegebenen Adresse, wenn das V-Flag gesetzt ist (Ergebnis der letzten Operation ergab einen Überlauf von Bit 6 in Bit 7).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | 70 | 2 | 2 |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: JMP

*Funktion*:
Verzweigt zur absolut oder indirekt angegebenen Adresse.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | 4C | 3 | 3 |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | 6C | 3 | 5 |

Beeinflußte Flags keine

*Befehl*: JSR

*Funktion*:
Verzweigt zur absolut angegebenen Adresse, wobei der aktuelle Inhalt des Programmzählers (+2) auf den Stack gebracht wird (Unterprogrammaufruf).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | 20 | 3 | 6 |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: RTS

*Funktion*:
Verzweigt zu jener Adresse, die von JSR auf den Stack gebracht wurde (Rückkehr vom Unterprogramm). Achtung: der Stapelzeiger muß vor RTS exakt den gleichen Zustand wie vor dem Unterprogrammaufruf mit JSR besitzen. Im Unterprogramm muß daher Befehlen, die Elemente auf den Stack bringen, eine gleichartige Anzahl von Befehlen folgen, die Elemente vom Stack ziehen!

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 60 | 1 | 6 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

## 28.5. Arithmetik-Befehle

*Befehl*: CLC

*Funktion*:
Löscht das Carry-Flag (Vorbereitung einer Addition).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 18 | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | | | | | | | X |

*Befehl*: SEC

*Funktion*:
Setzt das Carry-Flag (Vorbereitung einer Subtraktion).

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 38 | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |

```
Zeropage-X-indiziert:
Zeropage-Y-indiziert:
Relativ:
Indirekt:

Beeinflußte Flags N-Flag  V-Flag  B-Flag  D-Flag  I-Flag  Z-Flag  C-Flag
                                                                    X
```

*Befehl*: CLV

*Funktion*:
Löscht das V-Flag.

```
Adressierungsarten      Code   Bytes   Taktzyklen

Implizit:               B8       1         2
Absolut:
Zeropage:
Unmittelbar:
X-indiziert:
Y-indiziert:
indirekt-X-indiziert:
indirekt-Y-indiziert:
Zeropage-X-indiziert:
Zeropage-Y-indiziert:
Relativ:
Indirekt:

Beeinflußte Flags N-Flag  V-Flag  B-Flag  D-Flag  I-Flag  Z-Flag  C-Flag
                            X
```

*Befehl*: CLD

*Funktion*:
Löscht das D-Flag (Dezimal-Flag), um nachfolgende Arithmetik-Befehle binär durchzuführen.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | D8 | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | | | | X | | | |

*Befehl*: SED

*Funktion*:
Setzt das D-Flag (Dezimal-Flag), um nachfolgende Arithmetik-Befehle dezimal durchzuführen.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | F8 | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | | | | X | | | |

*Befehl*: ADC

*Funktion*:
Addiert zum Inhalt des Akkumulators einen angegebenen Wert oder den Inhalt der adressierten Speicherzelle plus den Inhalt des Carry-Bits. Der Übertrag (0 oder 1) kommt anschließend in das Carry-Bit.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | 6D | 3 | 4 |
| Zeropage: | 65 | 2 | 3 |
| Unmittelbar: | 69 | 2 | 2 |
| X-indiziert: | 7D | 3 | 4 |
| Y-indiziert: | 79 | 3 | 4 |
| indirekt-X-indiziert: | 61 | 2 | 6 |
| indirekt-Y-indiziert: | 71 | 2 | 5 |
| Zeropage-X-indiziert: | 75 | 2 | 4 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | X | | | | X | X |

*Befehl*: SBC

*Funktion*:
Subtrahiert vom Inhalt des Akkumulators einen angegebenen Wert oder den Inhalt der adressierten Speicherzelle und den komplementierten (umgekehrten) Inhalt des Carry-Bits. Der komplementierte Übertrag (0 oder 1) kommt anschließend in das Carry-Bit.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | ED | 3 | 4 |
| Zeropage: | E5 | 2 | 3 |
| Unmittelbar: | E9 | 2 | 2 |
| X-indiziert: | FD | 3 | 4 |
| Y-indiziert: | F9 | 3 | 4 |
| indirekt-X-indiziert: | E1 | 2 | 6 |
| indirekt-Y-indiziert: | F1 | 2 | 5 |
| Zeropage-X-indiziert: | F5 | 2 | 4 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | X | | | | X | X |

## 28.6. Logische Operationen

*Befehl*: AND

*Funktion*:
UND-Verknüpfung des Akkumulator-Inhaltes mit dem angegebenen Wert oder dem Inhalt der adressierten Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | 2D | 3 | 4 |
| Zeropage: | 25 | 2 | 3 |
| Unmittelbar: | 29 | 2 | 2 |
| X-indiziert: | 3D | 3 | 4 |
| Y-indiziert: | 39 | 3 | 4 |
| indirekt-X-indiziert: | 21 | 2 | 6 |
| indirekt-Y-indiziert: | 31 | 2 | 5 |
| Zeropage-X-indiziert: | 35 | 2 | 4 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: ORA

*Funktion*:
ODER-Verknüpfung des Akkumulator-Inhaltes mit dem angegebenen Wert oder dem Inhalt der adressierten Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | 0D | 3 | 4 |
| Zeropage: | 05 | 2 | 3 |
| Unmittelbar: | 09 | 2 | 2 |
| X-indiziert: | 1D | 3 | 4 |

| | | | |
|---|---|---|---|
| Y-indiziert: | 19 | 3 | 4 |
| indirekt-X-indiziert: | 01 | 2 | 6 |
| indirekt-Y-indiziert: | 11 | 2 | 5 |
| Zeropage-X-indiziert: | 15 | 2 | 4 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

*Befehl*: EOR

*Funktion*:
EXCLUSIC-ODER-Verknüpfung des Akkumulator-Inhaltes mit dem angegebenen Wert oder dem Inhalt der adressierten Speicherzelle.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | 4D | 3 | 4 |
| Zeropage: | 45 | 2 | 3 |
| Unmittelbar: | 49 | 2 | 2 |
| X-indiziert: | 5D | 3 | 4 |
| Y-indiziert: | 59 | 3 | 4 |
| indirekt-X-indiziert: | 41 | 2 | 6 |
| indirekt-Y-indiziert: | 51 | 2 | 5 |
| Zeropage-X-indiziert: | 55 | 2 | 4 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | |

## 28.7. Schiebe- und Rotierbefehle

*Befehl*: ASL

*Funktion*:
Nach links schieben des Akkumulator-Inhaltes oder des Inhaltes der adressierten Speicherzelle. Bit 7 wird in das Carry-Bit geschoben, als neues Bit 0 eine null nachgeschoben.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Akkumulator: | 0A | 1 | 2 |
| Absolut: | 0E | 3 | 6 |
| Zeropage: | 06 | 2 | 5 |
| Unmittelbar: | | | |
| X-indiziert: | 1E | 3 | 7 |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | 16 | 2 | 6 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | X |

*Befehl*: LSR

*Funktion*:
Nach rechts schieben des Akkumulator-Inhaltes oder des Inhaltes der adressierten Speicherzelle. Bit 0 wird in das Carry-Bit geschoben, als neues Bit 7 eine null nachgeschoben.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Akkumulator: | 1A | 1 | 2 |
| Absolut: | 4E | 3 | 6 |
| Zeropage: | 46 | 2 | 5 |
| Unmittelbar: | | | |
| X-indiziert: | 5E | 3 | 7 |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | 56 | 2 | 6 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | X |

*Befehl*: ROL

*Funktion*:
Nach links rotieren des Akkumulator-Inhaltes oder des Inhaltes der adressierten Speicherzelle. Bit 7 wird in das Carry-Bit geschoben, als neues Bit 0 der aktuelle Inhalt des Carry-Bits nachgeschoben.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Akkumulator: | 2A | 1 | 2 |
| Absolut: | 2E | 3 | 6 |
| Zeropage: | 26 | 2 | 5 |
| Unmittelbar: | | | |
| X-indiziert: | 3E | 3 | 7 |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |

| | | | |
|---|---|---|---|
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | 36 | 2 | 6 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | X |

*Befehl*: ROR

*Funktion*:
Nach rechts rotieren des Akkumulator-Inhaltes oder des Inhaltes der adressierten Speicherzelle. Bit 0 wird in das Carry-Bit geschoben, als neues Bit 7 der aktuelle Inhalt des Carry-Bits nachgeschoben.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Akkumulator: | 6A | 1 | 2 |
| Absolut: | 6E | 3 | 6 |
| Zeropage: | 66 | 2 | 5 |
| Unmittelbar: | | | |
| X-indiziert: | 7E | 3 | 7 |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | 76 | 2 | 6 |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | | | | | X | X |

## 28.8. Unterbrechungsbefehle

*Befehl*: BRK

*Funktion*:
Software-Unterbrechung. Der Programmzähler und das Status-Register werden auf den Stack gebracht und das B-Flag (BRK-Flag) gesetzt. Anschließend erfolgt ein Sprung über den Vektor $FFFE/$FFFF.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 00 | 1 | 7 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: RTI

*Funktion*:
Rückkehr von einer Unterbrechung zu jener Adresse, die vor der Unterbrechung auf den Stack gebracht wurde. Das durch die Unterbrechung auf den Stack gebrachte Status-Register wird durch RTI wiederhergestellt.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 40 | 1 | 6 |
| Absolut: | | | |

```
Zeropage:
Unmittelbar:
X-indiziert:
Y-indiziert:
indirekt-X-indiziert:
indirekt-Y-indiziert:
Zeropage-X-indiziert:
Zeropage-Y-indiziert:
Relativ:
Indirekt:

Beeinflußte Flags N-Flag  V-Flag  B-Flag  D-Flag  I-Flag  Z-Flag  C-Flag
                    X       X       X       X       X       X       X
```

*Befehl*: SEI

*Funktion*:
Setzt das I-Flag (Interrupt-Flag) im Status-Register. Nachfolgende Unterbrechungen werden dadurch verhindert.

```
Adressierungsarten      Code   Bytes   Taktzyklen

Implizit:               78     1       2
Absolut:
Zeropage:
Unmittelbar:
X-indiziert:
Y-indiziert:
indirekt-X-indiziert:
indirekt-Y-indiziert:
Zeropage-X-indiziert:
Zeropage-Y-indiziert:
Relativ:
Indirekt:

Beeinflußte Flags N-Flag  V-Flag  B-Flag  D-Flag  I-Flag  Z-Flag  C-Flag
                                                                    X
```

*Befehl*: CLI

*Funktion*:
Löscht das I-Flag (Interrupt-Flag) im Status-Register. Nachfolgende Unterbrechungen werden dadurch wieder zugelassen.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | 58 | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | | | | | X | | |

## 28.9. Sonstige Befehle

*Befehl*: NOP

*Funktion*:
NOP verändert weder Speicherzellen noch Register (also auch nicht das Status-Register) und wird daher in Warteschleifen oder als Platzhalter für später in das Programm einzufügende Befehle verwendet.

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | EA | 1 | 2 |
| Absolut: | | | |
| Zeropage: | | | |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

Beeinflußte Flags keine

*Befehl*: BIT

*Funktion*:
Mit dem BIT-Befehl können gezielt Bits einer adressierten Speicherzelle getestet werden. Der Inhalt des Akkumulators wird mit dem Inhalt der angegebenen Speicherzelle UND-verknüpft. Ist das Ergebnis null, wird das Zero-Flag gesetzt, ansonsten gelöscht. Zusätzlich werden die Bits 6 und 7 der adressierten

Speicherzelle in das V-Flag und das N-Flag kopiert, die anschließend ebenfalls getestet werden können. Der Inhalt des Akkumulators wird nach der Verknüpfung original wiederhergestellt!

| Adressierungsarten | Code | Bytes | Taktzyklen |
|---|---|---|---|
| Implizit: | | | |
| Absolut: | 2C | 3 | 4 |
| Zeropage: | 24 | 2 | 3 |
| Unmittelbar: | | | |
| X-indiziert: | | | |
| Y-indiziert: | | | |
| indirekt-X-indiziert: | | | |
| indirekt-Y-indiziert: | | | |
| Zeropage-X-indiziert: | | | |
| Zeropage-Y-indiziert: | | | |
| Relativ: | | | |
| Indirekt: | | | |

| Beeinflußte Flags | N-Flag | V-Flag | B-Flag | D-Flag | I-Flag | Z-Flag | C-Flag |
|---|---|---|---|---|---|---|---|
| | X | X | | | | X | |

## 29. Die Routinen des Betriebssystems

Der folgende Abschnitt führt alle besprochenen Betriebssystem-Routinen auf. Die Adressen dieser Routinen sind bei allen Commodore-Rechnern identisch!

*Routine*: BSOUT ($FFD2)

*Funktion*:
Ausgabe eines Zeichens auf das Standardgerät Bildschirm oder eine geöffnete logische Datei, auf die zuvor mit CHKOUT die Ausgabe umgelenkt wurde.

```
Parameter hin: ASCII-Code des Zeichens im Akkumulator.
Parameter zurück: keine
Beeinflußte Register: keine
```

*Routine*: BASIN ($FFCF)

*Funktion*:
Eingabe eines Zeichens vom Standardgerät Tastatur oder einer geöffneten logischen Datei, auf die zuvor mit CHKIN die Eingabe umgelenkt wurde.

```
Parameter hin: keine
Parameter zurück: Zeichen im Akkumulator (ASCII-Code).
Beeinflußte Register: Akkumulator, X-Register
```

*Routine*: GETIN ($FFE4)

*Funktion*:
Zeichen von der Tastatur lesen.

Parameter hin: keine
Parameter zurück: Zeichen im Akkumulator (ASCII-Code). $00, wenn keine Taste gedrückt.
Beeinflußte Register: Akkumulator, X-Register, Y-Register

*Routine*: CHKOUT ($FFC9)

*Funktion*: Leitet die Ausgabe auf eine zuvor geöffnete logische Datei um.

Parameter hin: Filenummer (Dateinummer) im X-Register.
Parameter zurück: keine
Beeinflußte Register: Akkumulator, X-Register

*Routine*: CHKIN ($FFC6)

*Funktion*:
Leitet die Eingabe auf eine zuvor geöffnete logische Datei um.

Parameter hin: Filenummer (Dateinummer) im X-Register.
Parameter zurück: keine
Beeinflußte Register: Akkumulator, X-Register

*Routine*: CLRCH ($FFCC)

*Funktion*:
Schließt alle geöffneten Datenübertragungskanäle (ersetzt jedoch nicht das folgende Schliessen der logischen Datei mit CLOSE) und lenkt die Ein-/Ausgabe auf die Standardgeräte Bildschirm und Tastatur um.

Parameter hin: keine
Parameter zurück: keine
Beeinflußte Register: Akkumulator, X-Register

*Routine*: PLOT ($FFF0)

*Funktion*:
Setzt den Cursor auf die angegebene Position oder holt die aktuelle Cursorposition.

Parameter hin: Cursor setzen: Carry-Flag löschen, Spalte im Y-Register (0-39), Zeile im X-Register (0-24). Cursorposition holen: Carry-Flag setzen.
Parameter zurück: Cursorposition holen: Spalte im Y-Register, Zeile im X-Register.
Beeinflußte Register: Akkumulator, X-Register, Y-Register

## 30. Die Routinen des BASIC-Interpreters

Die Routinen des BASIC-Interpreters unterscheiden sich leider von Rechner zu Rechner. Die Funktionsweise ist jedoch immer identisch, ebenso wie die hin- bzw. zurückübergebenen Parameter.

Sie dürfen davon ausgehen, daß jede der im folgenden genannten Routinen alle Register beeinflußt!

*Routine*: CHKKOM

*Adresse*: C64:$AEFD; C16, Plus/4:$9491; C128:$795C

*Funktion*:
Liest das aktuelle Zeichen aus dem BASIC-Text und prüft, ob es sich um ein Komma handelt.

```
Parameter hin: keine
Parameter zurück: keine
```

*Routine*: GETBYT

*Adresse*: C64:$B79E; C16, Plus/4:$9D84; C128:$87F4

*Funktion*:
Liest einen Ein-Byte-Wert aus dem BASIC-Text oder eine numerische Variable mit entsprechendem Inhalt (0-255).

```
Parameter hin: keine
Parameter zurück: Byte-Wert im X-Register.
```

*Routine*: FRMNUM

*Adresse*: C64:$AD8A; C16, Plus/4:$9314; C128:$FFD7

*Funktion*:
Wertet einen beliebigen numerischen Ausdruck aus, der sich im BASIC-Text befindet.

```
Parameter hin: keine
Parameter zurück: Ergebnis im Fließkomma-Format im FAC (Fließkomma-Akkumulator).
```

*Routine*: ADRFOR

*Adresse*: C64:$B7F7; C16, Plus/4:$9DE4; C128:$8815

*Funktion*:
Wandelt eine Fließkomma-Zahl, die sich im FAC befindet, ins Adreßformat (Low-Byte/High-Byte) und kann sehr effektiv zusammen mit FRMNUM zum Einlesen eines Zwei-Byte-Wertes aus dem BASIC-Text eingesetzt werden.

```
Parameter hin: Fließkomma-Zahl im FAC.
Parameter zurück: Integerzahl im Adreßformat im Y-Register (Low-Byte) und im Akkumulator (High-Byte).
```

*Routine*: ADRBYT

*Adresse*: C64:$B7EB; C16, Plus/4:$9DD2; C128:$8803

*Funktion*:
Liest einen Zwei-Byte-Wert und einen folgenden - durch ein Komma getrennten - Ein-Byte-Wert aus dem BASIC-Text ein.

```
Parameter hin: keine
Parameter zurück: Zwei-Byte-Wert im Adreßformat in $14/$15 und Ein-Byte-Wert im X-Register.
```

*Routine*: GETPOS

*Adresse*: C64:$B08B; C16, Plus/4:$96A5; C128:$7AAF

*Funktion*:
Liest eine Variable beliebigen Typs aus dem BASIC-Text und übergibt einen Zeiger auf die Variable (numerische Variablen) bzw. auf die Descriptoren (Stringvariablen).

```
Parameter hin: keine
Parameter zurück: Zeiger in Akkumulator (Low-Byte) und im Y-Register
(High-Byte).
```

*Routine*: STRRES

*Adresse*: C64:$B4F4; C16, Plus/4:$A906; C128:$9299

*Funktion*:
Reserviert Platz für einen anzulegenden String und vermindert zugleich den Zeiger auf das Ende des String-Stacks (FRETOP: C64/C16, Plus/4:$33/$34, C128:$35/$36) um die Stringlänge (+2 beim C16, Plus/4/C128 bedingt durch zwei Bytes für den Rückzeiger). Ist nicht ausreichend Platz vorhanden, gibt STRRES die Fehlermeldung OUT OF MEMORY ERROR aus.

```
Parameter hin: Stringlänge im Akkumulator
Parameter zurück: keine
```

*Routine*: INTOUT

*Adresse*: C64:$BDCD; C16, Plus/4:$A45F; C128:$8E32

*Funktion*:
Gibt eine Integerzahl ab der aktuellen Cursorposition auf dem Bildschirm aus.

```
Parameter hin: Integerzahl im X-Register (Low-Byte) und Akkumulator
(High-Byte).
Parameter zurück: keine
```

## 31. Der Maschinensprache-Monitor

Der hier erklärte und im Anhang auch zum Abtippen aufgeführte Maschinensprache-Monitor verfügt über Möglichkeiten, die über den durchschnittlichen Standard hinausgehen. Aus diesem Grund könnte er auch für diejenigen unter Ihnen interessant sein, die bereits einen Maschinensprache-Monitor besitzen.

Der abgedruckte Monitor liegt ab $9000 (36834). Er besitzt keinen Assemblerbefehl A. Stattdessen ändern Sie einfach die disassemblierten Assemblerbefehle oder setzen ein Komma, gefolgt von der gewünschten Startadresse und dem Befehl. Um also ein Programm ab $C000 einzugeben das mit LDA $0400 beginnt geben Sie ein:

```
, C000 LDA $0400
```

Mit diesem Monitor ist es möglich, jede beliebige Speicherkonfiguration einzuschalten. Somit ist es möglich, auch die Speicherbereiche 'unter' dem BASIC-ROM, dem KERNAL-ROM und 'unter' dem I/O-Bereich auszulesen und zu verändern. Selbstverständlich können in diesen Bereichen auch Programme gestartet werden. Eine Besonderheit dieses Monitors ist, außer einigen verbesserten Standardbefehlen, die Möglichkeit, Daten aus den eben erwähnten Bereichen mit Hilfe veränderter LOAD- und SAVE-Routinen zu laden und auch von dort zu speichern.

Zu beachten ist noch, daß alle Eingaben mit RETURN abgeschlossen werden müssen.

Jetzt werden wir näher auf die einzelnen Befehle eingehen:

C

COMPARE: Vergleichen von zwei Speicherbereichen. Nicht übereinstimmende Speicherzellen werden angezeigt.

```
Format:  C 1000 1800 2000
```

Der Bereich von $1000 bis $1800 wird mit dem Bereich ab $2000 verglichen.

D

Disassemblieren: Disassembliert den angegebenen Speicherbereich. Beim Schreiben eigener Programme brauchen Sie nur den bestehenden Befehl zu überschreiben.

```
Format:  D 1000 1100
```

Disassembliert den Speicherbereich von $1000 bis $1100. Wenn keine Bereichsbegrenzung angeben wird, wird nur die erste Speicherzelle angegeben.

F

Fill: Füllt den angegebenen Speicherbereich mit den nach der Speicherbereichangabe stehenden Bytes.

```
Format:  F 1000 2000 46 49 4C 4C
```

Der Speicherbereich von $1000 bis $2000 wird mit den Bytes $46, $49, $4C, $4C gefüllt, was dem Wort 'FILL' im ASCII-Code entspricht.

G

Goto start a Program: Startet ein Programm ab der angegebenen Adresse.

```
Format:   G C000
```

Das Programm wird ab der Adresse $C000 gestartet.

H

Hunt: Durchsucht den angegebenen Speicherbereich nach der angegebenen Bytefolge. Es ist auch möglich, nach teilweise unbekannten Bytefolgen zu suchen.

```
Format:   H E000 FFFF 4C ?? FF
```

Durchsucht den Speicherbereich von E000 bis FFFF. Die Fragezeichen stehen für ein unbekanntes Byte.

L

Load: Lädt ein Programm in den Speicher.

```
Format:   L "FILENAME"
```

Lädt ein Progamm absolut von Disk in den Speicher.

```
Format:   L "FILENAME",08,1000
```

Lädt ein Programm von Disk ab der Adresse $1000 in den Speicher. Die auf Disk angegebene Adresse wird ignoriert. Die 08 ist die Geräteadresse. Soll ein Programm von Kassette geladen werden, so muß auf das zweite Anführungszeichen eine 01 folgen.

```
Format:   L"FILENAME",07,D000
```

Lädt ein Programm von Disk immer in den RAM-Bereich des Rechners. Bei dieser Anweisung wird demnach nicht wie normalerweise der I/O-Bereich, sondern der darunterliegende RAM-Bereich beschrieben. Wenn keine Angabe der Startadresse erfolgt, wird das Programm absolut geladen. Diese Anweisung ist nur von Disk möglich.

M

Memory-Dump: Zeigt die Speicherzellen des angegebenen Speicherbereichs und zusätzlich deren Umsetzung in Bildschirmzeichen an.

```
Format:  M 1000 2000
```

Zeigt den Speicherbereich von $1000 bis $2000 an.

P

Printer: Nach der Eingabe von P werden alle Ausgaben anstatt auf dem Bildschirm auf dem Drucker mit der Geräteadresse 4 ausgegeben. Nach erneuter Eingabe von P erfolgt die Ausgabe wieder auf den Bildschirm.

```
Format:  P
```

R

Register: Zeigt die letzten Registerwerte an.

PC Programmzeiger
IRQ Interruptvektor
SR Prozessorstatusregister
AC Akkumulator
XR X-Register
YR Y-Register
SP Stapelzeiger

```
Format:  R
```

Die Register werden nach einem BREAK automatisch angezeigt.

S

SAVE: Speichert den angegebenen Bereich.

```
Format:  S"FILENAME",08,1000,2000
```

Speichert den Bereich von $1000 bis $2000 auf das Gerät mit der Geräteadresse 08 (Disk). Für das Speichern auf Kassette muß Geräteadresse 01 gewählt werden.

```
Format:  S"FILENAME",07,E000,FFFF
```

Speichert den angegebenen Bereich je nach Speicherkonfiguration auf Diskette ab. Somit ist es möglich, den RAM-Bereich unter Kernal-ROM, I/O-Bereich und BASIC-ROM abzuspeichern. Diese Funktion existiert nur im Zusammenhang mit dem Gebrauch einer Diskettenstation.

T

Transfer: Kopiert den angegebenen Speicherbereich in einen anderen angegebenen Speicherbereich.

```
Format: T 1000 2000 1800
```

Kopiert den Bereich von $1000 bis $2000 nach $1800. Bei Bereichsüberschreitungen treten beim Kopieren, sowohl nach oben als auch nach unten, keine Fehler auf.

X

EXIT: Rücksprung ins BASIC.

```
Format:  X
```

Disk Command: Gibt den Status der Floppy auf den Bildschirm aus.

```
Format:  @
```

Erfolgen auf das @ weitere Zeichen, so entspricht dieses einem OPEN 1,8,15," wobei die nachfolgenden Zeichen an die Floppy übergeben werden.

```
Format:  @I
```

@$

```
Format:  @$
```

Gibt die Directory auf den Bildschirm aus.

+

Addition: Addiert zwei hexadezimale Zahlen. Diese Zahlen müssen immer vierstellig angegeben werden.

```
Format:  + 2000 0152
```

Addiert $2000 mit $0152 und gibt das Ergebnis aus.

-

Subtraktion: Subtrahiert zwei Hexadezimale Zahlen.

```
Format:   - 2000 0152
```

Subtrahiert $0152 von $2000 und gibt das Ergebnis aus.

$

Umrechnung vom Hexadezimalsystem ins Dezimalsystem.

```
Format:   $1000
```

#

Umrechnung vom Dezimalsystem ins Hexadezimalsystem.

```
Format:   #4096
```

Anmerkung: Fast alle Befehle lassen sich durch das Betätigen der RUN/STOP-Taste unterbrechen.

Wenn Sie das Betriebssystem vom Monitor aus ausgeschaltet haben und ein Programm starten, das mit BRK endet, so wird sich der Rechner unweigerlich 'aufhängen'. Das gleiche geschieht, wenn Sie mit ausgeschaltetem BASIC oder Betriebssystem den Monitor mit X verlassen.

Durch die Möglichkeit, eine beliebige Speicherkonfiguration einzustellen, läßt sich auch das Zeichensatz-ROM auslesen, kopieren und saven.

Der folgende BASIC Loader erzeugt den beschriebenen Maschinensprachemonitor, der im Speicherbereich $9000 (36864) bis $9D10 (40208) liegt. Speichern Sie Ihn bitte nach dem Abtippen undbedingt ab. Wenn der Monitor fehlerfrei läuft, können Sie Ihn auch von sich selbst aus als Maschinensprache-Programm abspeichern. Er ist dann wesentlich kürzer und somit schneller zu laden. Nach dem Laden des so abgespeicherten Monitors setzen Sie die BASIC-Zeiger mit NEW zurück und starten ihn mit

```
SYS 9*4096
```

```
5 N=9*16^3 1
10 READ X:IF X=-1 THEN 30:POKE N,X
15 S=S+X:N=N+1:GOTO 10
30 IF S<>363070 OR N<>40208 THEN PRINT"FEHLER IN DATAS":END
35 SYS 9*16^3
36 DATA120,169,39,141,22,3,169,152,141,23,3,169,0,133,103,76,139,152,162
37 DATA255,160,155,132,99,160,0,240,11,169,168,133,98,32,17,145,32,141,1
45
38 DATA96,177,96,232,224,160,240,238,221,171,155,208,246,138,56,249,216
39 DATA156,144,3,200,208,247,132,2,152,10,24,101,2,133,98,32,17,145,189
40 DATA65,156,208,8,169,35,32,240,145,76,95,145,201,1,208,3,76,113,145,2
01
41 DATA2,208,6,32,113,145,76,156,145,201,3,208,6,32,113,145,76,166,145,2
01
42 DATA4,208,3,76,95,145,201,5,208,6,32,95,145,76,156,145,201,6,208,6,32
43 DATA95,145,76,166,145,201,7,208,16,169,40,32,240,145,32,95,145,32,156
44 DATA145,169,41,76,240,145,201,8,208,16,169,40,32,240,145,32,95,145,16
9
45 DATA41,32,240,145,76,166,145,201,9,208,58,32,141,145,169,36,32,240,14
5
46 DATA160,0,177,96,48,28,56,101,96,72,144,10,164,97,200,152,32,58,145,2
4
47 DATA144,5,165,97,32,58,145,104,32,58,145,76,141,145,56,101,96,72,176
48 DATA238,164,97,136,152,32,58,145,24,144,233,201,10,208,13,169,40,32,2
40
49 DATA145,32,113,145,169,41,76,240,145,76,141,145,32,176,145,169,58,32
50 DATA240,145,76,48,145,32,176,145,169,44,32,240,145,32,48,145,32,47,14
6
51 DATA160,0,177,98,32,240,145,200,192,3,208,246,169,32,76,240,145,165,9
7
52 DATA32,58,145,165,96,76,58,145,32,70,145,72,152,32,240,145,104,76,240
```

```
53 DATA145,72,74,74,74,74,32,83,145,168,104,76,83,145,41,15,201,10,24,48
54 DATA2,105,7,105,48,96,169,36,32,240,145,32,141,145,160,0,177,96,32,58
55 DATA145,76,141,145,169,36,32,240,145,160,2,177,96,32,58,145,160,1,177
56 DATA96,32,58,145,32,141,145,32,141,145,76,141,145,230,96,208,10,230,9
7
57 DATA208,6,72,169,255,133,102,104,96,169,44,32,240,145,169,88,76,240,1
45
58 DATA169,44,32,240,145,169,89,76,240,145,32,42,146,169,46,76,240,145,3
2
59 DATA200,145,133,2,152,32,200,145,10,10,10,10,5,2,96,201,65,56,48,2,23
3
60 DATA7,233,48,96,32,217,145,170,76,217,145,32,231,145,201,32,240,249,1
68
61 DATA32,231,145,76,184,145,32,52,146,240,1,96,76,64,146,32,11,146,32,2
10
62 DATA255,76,23,146,32,11,146,32,207,255,76,23,146,32,11,146,32,62,241
63 DATA76,23,146,72,165,1,133,251,169,55,133,1,104,88,96,72,120,165,251
64 DATA133,1,104,96,32,2,146,201,3,240,1,96,76,64,146,169,13,76,240,145
65 DATA169,32,76,240,145,32,249,145,201,13,96,32,231,145,201,32,96,32,17
6
66 DATA145,162,255,154,32,52,146,240,245,201,32,240,244,201,46,240,240,1
60
67 DATA19,136,48,14,217,138,154,208,248,185,100,154,72,185,119,154,72,96
68 DATA169,63,32,240,145,76,64,146,32,158,146,32,31,146,32,127,146,144,1
99
69 DATA32,18,144,76,113,146,165,102,240,7,169,0,133,102,76,154,146,165,9
7
70 DATA197,101,144,12,208,8,165,96,197,100,144,4,240,2,24,96,56,96,32,21
0
71 DATA145,134,97,133,96,32,52,146,240,16,201,32,208,184,32,220,145,170
72 DATA32,217,145,134,101,133,100,96,165,96,133,100,165,97,133,101,96,32
73 DATA210,145,134,97,133,96,32,52,146,208,151,88,108,96,0,120,108,2,160
74 DATA32,158,146,32,31,146,32,127,146,144,6,32,235,146,76,218,146,76,64
75 DATA146,32,6,145,160,0,132,3,177,96,72,32,47,146,104,32,58,145,164,3
76 DATA200,192,8,208,237,162,8,160,0,32,47,146,169,1,133,199,177,96,41,1
27
77 DATA201,32,144,10,32,240,145,169,0,133,212,24,144,3,32,179,145,32,141
78 DATA145,202,208,229,169,0,133,199,96,32,210,145,134,97,133,96,160,0,3
2
79 DATA231,145,132,3,32,231,145,32,135,147,168,32,231,145,32,135,147,32
80 DATA184,145,164,3,145,96,200,192,8,208,226,169,145,32,240,145,32,235
81 DATA146,32,176,145,169,58,141,119,2,165,97,32,70,145,140,120,2,141,12
1
82 DATA2,165,96,32,70,145,140,122,2,141,123,2,169,32,141,124,2,169,6,133
83 DATA198,76,67,146,201,32,240,1,96,76,102,146,32,210,145,134,97,133,96
```

```
84 DATA169,155,133,99,32,58,146,240,251,32,62,148,224,31,144,11,224,39,1
76
85 DATA7,169,0,141,1,2,240,5,169,1,141,1,2,32,117,148,133,3,169,0,224,0
86 DATA240,12,202,24,125,216,156,76,189,147,224,0,208,240,164,4,190,216
87 DATA156,168,165,3,217,65,156,208,10,185,171,155,160,0,145,96,24,144,7
88 DATA200,202,208,237,76,102,146,200,162,6,165,3,202,48,25,221,157,154
89 DATA208,248,165,98,145,96,169,145,32,240,145,32,18,144,32,176,145,169
90 DATA44,76,98,147,201,11,240,236,201,9,208,30,56,165,99,233,2,133,99,1
65
91 DATA98,229,97,201,2,144,4,201,255,208,75,32,114,149,24,101,99,145,96
92 DATA76,252,147,165,99,145,96,200,165,98,145,96,76,252,147,162,0,157,2
24
93 DATA158,232,224,3,240,6,32,231,145,24,144,242,162,0,134,98,160,255,20
0
94 DATA185,224,158,209,98,240,15,232,230,98,230,98,230,98,164,98,192,168
95 DATA176,9,144,231,192,2,208,229,134,4,96,76,102,146,32,52,146,208,3,1
69
96 DATA11,96,201,32,240,244,201,35,208,19,32,231,145,32,73,149,32,217,14
5
97 DATA133,98,32,52,146,208,221,169,0,96,201,40,208,54,32,231,145,32,73
98 DATA149,32,217,145,133,98,32,231,145,201,41,240,25,201,44,240,3,76,53
99 DATA149,32,87,149,32,231,145,201,41,208,179,32,52,146,208,174,169,7,9
6
100 DATA32,95,149,32,52,146,208,163,169,8,96,32,73,149,32,217,145,133,98
101 DATA32,52,146,240,83,201,44,240,52,32,224,145,133,99,32,52,146,240,3
1
102 DATA201,44,208,129,32,231,145,201,88,240,12,201,89,208,193,32,52,146
103 DATA208,188,169,3,96,32,52,146,208,180,169,2,96,173,1,2,240,3,169,1,
96
104 DATA169,9,96,32,231,145,201,88,240,12,201,89,208,47,32,52,146,208,42
105 DATA169,6,96,32,52,146,208,34,169,5,96,169,4,96,32,224,145,133,99,32
106 DATA231,145,201,41,208,16,32,52,146,208,11,169,10,96,201,36,240,3,76
107 DATA102,146,96,76,102,146,32,106,149,32,231,145,201,88,208,243,96,32
108 DATA106,149,32,231,145,201,89,208,232,96,32,231,145,201,44,208,224,9
6
109 DATA165,96,73,255,170,232,138,96,32,231,149,32,52,146,208,7,169,8,13
3
110 DATA186,76,174,149,201,44,208,12,32,217,145,133,186,32,52,146,240,23
111 DATA201,44,208,119,32,210,145,133,98,134,99,32,52,146,208,242,169,0,
133
112 DATA185,24,144,4,162,1,134,185,165,186,201,8,240,10,201,9,240,3,76,2
15
113 DATA149,32,21,150,169,0,32,11,146,166,98,164,99,32,213,255,32,105,15
0
114 DATA32,23,146,76,141,153,201,1,208,55,76,194,149,169,1,133,185,133,1
86
```

```
115 DATA76,194,149,32,52,146,240,242,201,32,240,247,201,34,208,30,162,0,
32
116 DATA249,145,201,34,240,10,157,224,158,232,224,17,240,13,208,239,134,
183
117 DATA169,158,133,188,169,224,133,187,96,76,102,146,169,165,162,244,32
118 DATA69,150,162,3,189,163,154,157,84,159,202,16,247,162,15,189,167,15
4
119 DATA157,105,159,202,16,247,162,2,189,183,154,157,142,159,202,16,247,
169
120 DATA3,141,138,159,198,186,96,32,11,146,133,100,134,101,160,0,177,100
121 DATA153,0,159,200,208,248,169,0,141,48,3,141,50,3,169,159,141,49,3,1
41
122 DATA51,3,76,23,146,162,3,189,76,253,157,48,3,202,16,247,96,32,231,14
9
123 DATA32,106,149,32,217,145,133,186,32,106,149,32,210,145,133,96,134,9
7
124 DATA32,106,149,32,210,145,133,98,134,99,32,52,146,208,53,165,186,201
125 DATA8,240,7,201,9,208,32,32,209,150,169,1,133,185,169,96,166,98,164,
99
126 DATA232,208,1,200,32,11,146,32,216,255,32,23,146,32,105,150,76,141,1
53
127 DATA72,32,42,146,104,201,1,208,2,240,216,76,102,146,169,237,162,245,
32
128 DATA69,150,162,2,189,186,154,157,52,159,202,16,247,162,4,189,189,154
129 DATA157,60,159,202,16,247,198,186,96,32,23,146,177,172,32,11,146,76,
221
130 DATA237,32,158,146,162,0,32,52,146,240,200,201,32,208,196,134,3,32,2
17
131 DATA145,166,3,157,0,159,232,32,52,146,208,236,134,3,162,0,32,127,146
132 DATA144,17,160,0,189,0,159,145,96,32,141,145,232,228,3,208,236,240,2
32
133 DATA76,64,146,32,158,146,32,210,145,133,98,134,99,228,97,144,6,208,2
7
134 DATA197,96,176,23,32,127,146,144,228,160,0,177,96,145,98,32,141,145,
230
135 DATA98,208,238,230,99,76,75,151,165,100,56,229,96,72,165,101,229,97,
144
136 DATA61,24,101,99,133,99,104,24,101,98,144,2,230,99,133,98,32,127,146
137 DATA144,204,160,0,177,100,145,98,198,100,165,100,201,255,208,2,198,1
01
138 DATA198,98,165,98,201,255,208,227,198,99,76,125,151,32,158,146,32,52
139 DATA146,240,125,162,0,201,32,208,119,32,247,151,144,16,134,3,32,224,
145
140 DATA166,3,157,0,159,232,32,52,146,208,231,32,42,146,134,3,160,0,32,1
27
```

```
141 DATA146,144,38,32,31,146,185,0,2,240,13,177,96,217,0,159,240,6,32,14
1
142 DATA145,76,199,151,200,196,3,208,225,32,48,145,32,141,145,32,47,146,
76
143 DATA199,151,76,64,146,32,231,145,201,63,208,29,32,249,145,201,63,208
144 DATA31,169,0,157,0,2,232,32,52,146,240,9,201,32,208,16,104,104,76,17
3
145 DATA151,24,96,72,169,255,157,0,2,104,56,96,76,102,146,120,216,104,14
1
146 DATA69,2,104,141,68,2,104,141,67,2,104,141,66,2,104,141,64,2,104,141
147 DATA65,2,206,64,2,208,3,206,65,2,186,142,70,2,162,0,189,194,154,32,2
40
148 DATA145,232,224,61,208,245,173,65,2,32,58,145,173,64,2,32,58,145,32,
47
149 DATA146,173,21,3,32,58,145,173,20,3,32,58,145,162,0,32,47,146,189,66
150 DATA2,32,58,145,232,224,5,208,242,32,42,146,76,64,146,141,67,2,142,6
8
151 DATA2,140,69,2,76,53,152,32,158,146,32,210,145,133,98,134,99,32,42,1
46
152 DATA32,127,146,144,29,160,0,177,96,209,98,208,12,32,141,145,230,98,2
08
153 DATA236,230,99,76,164,152,32,48,145,32,47,146,76,177,152,76,64,146,3
2
154 DATA210,145,168,32,42,146,138,72,152,170,104,32,11,146,32,205,189,32
155 DATA23,146,76,64,146,160,0,132,96,132,97,32,249,145,41,15,24,101,96,
133
156 DATA96,144,2,230,97,32,249,145,201,48,144,26,72,165,96,164,97,10,38,
97
157 DATA10,38,97,101,96,133,96,152,101,97,6,96,42,133,97,104,144,212,32,
42
158 DATA146,32,48,145,76,64,146,32,158,146,165,96,24,101,100,133,96,165,
97
159 DATA101,101,133,97,32,42,146,32,48,145,76,64,146,32,158,146,165,96,5
6
160 DATA229,100,133,96,165,97,229,101,133,97,76,47,153,32,11,146,32,231,
255
161 DATA32,23,146,32,52,146,240,52,201,32,240,247,201,36,240,86,162,0,15
7
162 DATA224,158,232,224,17,240,73,32,52,146,208,243,138,162,224,160,158,
32
163 DATA11,146,32,189,255,169,1,162,8,160,15,32,186,255,32,192,255,32,23
1
164 DATA255,32,23,146,32,42,146,32,11,146,169,8,133,186,32,180,255,169,1
11
165 DATA133,185,32,150,255,32,165,255,32,210,255,201,13,208,246,32,171,2
55
```

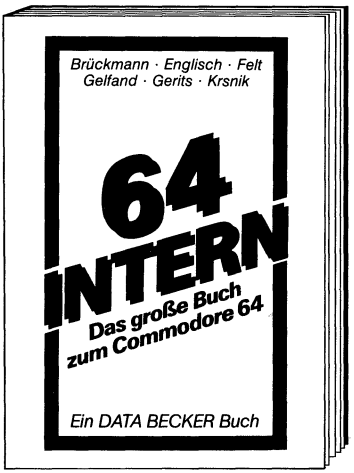
```
166 DATA32,23,146,76,64,146,76,102,146,72,32,42,146,104,32,11,146,141,22
4
167 DATA158,162,224,160,158,169,1,32,189,255,162,8,160,96,32,186,255,32,
213
168 DATA243,165,186,32,180,255,165,185,32,150,255,169,0,133,144,160,3,14
0
169 DATA224,158,32,165,255,141,225,158,164,144,208,49,32,165,255,164,144
170 DATA208,42,172,224,158,136,208,230,174,225,158,32,205,189,169,32,32,
210
171 DATA255,32,165,255,166,144,208,18,170,240,6,32,210,255,76,10,154,169
172 DATA13,32,210,255,160,2,208,194,32,66,246,32,23,146,76,64,146,32,52,
146
173 DATA208,48,32,42,146,32,11,146,165,103,208,27,198,103,169,1,162,4,32
174 DATA186,255,169,0,133,183,32,192,255,162,1,32,201,255,32,23,146,76,6
4
175 DATA146,230,103,32,231,255,32,204,255,76,80,154,76,102,146,146,146,1
46
176 DATA146,147,147,149,150,150,151,151,152,152,152,152,153,153,153,154,
109
177 DATA214,194,210,45,142,121,116,251,54,158,76,150,200,224,30,55,74,43
178 DATA68,77,71,88,58,44,76,83,70,84,72,82,67,36,35,43,45,64,80,0,4,5,6
179 DATA7,8,234,234,24,144,120,72,169,48,133,1,104,145,174,169,55,133,1,
234
180 DATA234,234,76,169,245,32,221,237,32,241,150,234,234,13,13,73,78,84,
69
181 DATA82,78,32,77,79,78,73,84,79,82,32,66,89,32,82,46,32,71,69,76,70,6
5
182 DATA78,68,13,13,32,32,32,80,82,32,32,73,82,81,32,32,83,82,32,65,67,3
2
183 DATA88,82,32,89,82,32,83,80,13,46,42,0,76,68,65,76,68,88,76,68,89,83
184 DATA84,65,83,84,88,83,84,89,84,65,88,84,65,89,84,88,65,84,89,65,84,8
8
185 DATA83,84,83,88,80,76,65,80,72,65,80,76,80,80,72,80,65,68,67,83,66,6
7
186 DATA73,78,67,68,69,67,73,78,88,68,69,88,73,78,89,68,69,89,65,78,68,7
9
187 DATA82,65,69,79,82,67,77,80,67,80,88,67,80,89,66,73,84,66,67,67,66,6
7
188 DATA83,66,69,81,66,78,69,66,77,73,66,80,76,66,86,67,66,86,83,74,77,8
0
189 DATA74,83,82,65,83,76,76,83,82,82,79,76,82,79,82,67,76,67,67,76,68,6
7
190 DATA76,73,67,76,86,83,69,67,83,69,68,83,69,73,78,79,80,82,84,83,82,8
4
191 DATA73,66,82,75,63,63,63,169,173,189,185,165,181,161,177,162,174,190
```

```
192 DATA166,182,160,172,188,164,180,141,157,153,133,149,129,145,142,134,
150
193 DATA140,132,148,170,168,138,152,154,186,104,72,40,8,105,109,125,121,
101
194 DATA117,97,113,233,237,253,249,229,245,225,241,238,254,230,246,206,2
22
195 DATA198,214,232,202,200,136,41,45,61,57,37,53,33,49,9,13,29,25,5,21,
1
196 DATA17,73,77,93,89,69,85,65,81,201,205,221,217,197,213,193,209,224,2
36
197 DATA228,192,204,196,44,36,144,176,240,208,48,16,80,112,76,108,32,14,
30
198 DATA6,22,10,78,94,70,86,74,46,62,38,54,42,110,126,102,118,106,24,216
199 DATA88,184,56,248,120,234,96,64,0,1,2,3,4,5,7,8,0,1,3,4,6,0,1,2,4,5,
1
200 DATA2,3,4,5,7,8,1,4,6,1,4,5,11,11,11,11,11,11,11,11,11,11,0,1,2,3,4,
5
201 DATA7,8,0,1,2,3,4,5,7,8,1,2,4,5,1,2,4,5,11,11,11,11,0,1,2,3,4,5,7,8,
0
202 DATA1,2,3,4,5,7,8,0,1,2,3,4,5,7,8,0,1,2,3,4,5,7,8,0,1,4,0,1,4,1,4,9,
9
203 DATA9,9,9,9,9,9,1,10,1,1,2,4,5,11,1,2,4,5,11,1,2,4,5,11,1,2,4,5,11,1
1
204 DATA11,11,11,11,11,11,11,11,11,11,8,5,5,7,3,3,1,1,1,1,1,1,1,1,1,8,
8
205 DATA4,4,1,1,1,1,8,8,8,8,3,3,2,1,1,1,1,1,1,1,1,2,1,5,5,5,5,1,1,1,1,1,
1
206 DATA1,1,1,1,1,-1
```

1 Das Zeichen "^" entspricht dem Hochpfeil auf der COMMODORE-Tastatur.

# 32. Index

64 Intern ist ein Standardwerk zum Commodore 64, das vom ausführlich dokumentierten ROM-Listing über die detaillierte Hardwarebeschreibung bis zu nützlichen BASIC-Erweiterungen alles enthält, was man zum professionellen Einsatz des Commodore 64 wissen muß.




Aus dem Inhalt:

- Speicherbelegungspläne
- Der Soundcontroller und seine Programmierung
- Die Handhabung des AD-Wandlers
- Der Videocontroller
- Programmierung von Farbe und Grafik
- Die Zeichengenerator-Schnittstelle
- Sprites
- Ein-/Ausgabesteuerung
- Timer und Echtzeituhr
- Joystickprogrammierung
- So arbeitet der BASIC-Interpreter
- Mathematische Routinen – selbst entwickelt
- Der serielle IEC-Bus
- Programmierung der RS-232
- Die Belegung der Zero-Page
- Der Commodore-64-Schaltplan

**Brückmann, Englisch, Felt, Gelfand, Krsnik, Gerits**
**64 Intern**
**Hardcover, 628 Seiten, DM 69,–**
***ISBN 3-89011-000-2***

# Bücher zum Commodore 128

Ein Standardwerk zum COMMODORE 128, das für jeden unentbehrlich ist, der tiefer in den COMMODORE 128 einsteigen will. Das gesamte Betriebssystem ist ausführlich und gründlich kommentiert, Grafik, Soundbausteine, Prozessor und Peripherieanschlüsse sind genauestens beschrieben. Ein Buch, das für den professionellen Programmierer sehr schnell unentbehrlich wird.

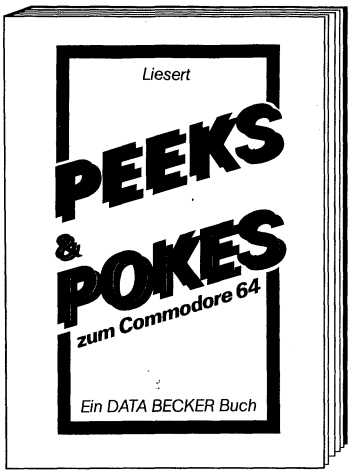

Aus dem Inhalt:

- Der VIC-Chip
  Registerbelegung
  Betriebsarten
  Zeichendarstellung, Graphik, Sprites und Soft-Scrolling
- Ein- und Ausgabesteuerung
  Die CIAs im COMMODORE 128
  Echtzeituhr
  Der serielle IEC-Bus des COMMODORE 128
- Der Sound-Chip SID
- Der 8563-VDC-Chip
  Pinbelegung
  Nutzung der VDC-Register
  Hires-Graphik mit 640×200 Punkten
- Das Memory-Management, die MMU
- Assemblerprogrammierung (Nutzung der Kernal-Routinen)
- Einbinden neuer BASIC-Befehle
- BASIC-Tokens
- Die CPU 8502
- Zeilenweise dokumentiertes Kernal-ROM
- ausführlich dokumentiertes BASIC 7.0
- Z-80-ROM dokumentiert (Boot-Sektor)
- Betriebssystem und Monitorlisting
- Die Hardware

**Schieb, Thrun, Wrobel**
**Commodore 128 Intern**
**Hardcover, 841 Seiten, DM 69,–**
***ISBN 3-89011-098-3***

## Bücher zum Commodore 64

Dieses Buch erklärt Ihnen leichtverständlich den Umgang mit Peeks & Pokes. Es enthält eine Beschreibung aller nutzbaren Speicheradressen und führt in die Hardware Ihres C64 ein. Die vielen sofort einsetzbaren Programme haben schnell dafür gesorgt, daß dieses Buch schon in der ersten Auflage zu einem unverzichtbaren Nachschlagewerk für jeden interessierten Programmierer wurde.



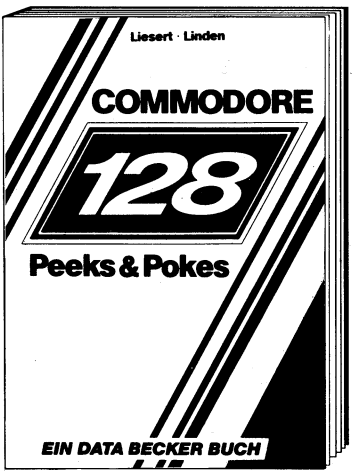

Aus dem Inhalt:

- Die Arbeitsweise der CPU
- Was ist ein Betriebssystem?
- Wie arbeitet der BASIC-Interpreter?
- Beschreibung und Nutzung der Zeropage
- Pointer & Stacks
- Speicherbelegungsplan
- Massenspeicher & Peripherie
- Die Spriteregister
- Programmierung der Schnittstellen
- Interruptprogrammierung
- Leichtverständliche Einführung in die Maschinensprache

**Liesert**
**Peeks & Pokes zum Commodore 64**
**196 Seiten, DM 29,–**
***ISBN 3-89011-032-0***

# Bücher zum Commodore 128

Schlagen Sie dem Betriebssystem Ihres C128 ein Schnippchen. Wie? Mit PEEKS & POKES natürlich! Dieses Buch erklärt leichtverständlich den Umgang damit. Mit einer riesigen Anzahl wichtiger POKES und ihren Anwendungsmöglichkeiten. Nebenbei wird der interne Aufbau Ihres neuen C128 prima erklärt.




Aus dem Inhalt:

- Die Arbeitsweise Ihres Rechners
- Was ist ein Betriebssystem?
- Wie arbeitet der Interpreter
- RAM-Erweiterungsbefehle
- Bankswitching
- Die Zeropage
- Pointer & Stacks
- Speicherbelegungsplan
- Massenspeicherung & Peripherie
- Der 40-/80-Zeichen-Bildschirm
- Sprites
- Grafik mit 640×200 Punkten
- Die Tastatur
- Der User-Port
- BASIC und Betriebssystem
- Grundlagen der Maschinensprache
- 8502-/Z80-Maschinensprache

**Liesert, Linden**
**Peeks & Pokes zum Commodore 128**
**248 Seiten, DM 29,–**
***ISBN 3-89011-138-6***

## *DAS STEHT DRIN:*

Endlich ist es soweit: Maschinensprache für Einsteiger ist das Buch, auf das alle, die sich schon immer für Maschinensprache interessierten, gewartet haben. Finden Sie heraus, was in Ihnen und in Ihrem Rechner steckt. Dieses Buch zeigt Ihnen, wie's geht: ohne Fachchinesisch, dafür einfach, schnell und effektiv. Nutzen Sie diese Chance.

Aus dem Inhalt:

- Einführung in Assembler
  Was man über Zahlen, Speicher und den Rechner wissen sollte
- Wie man mit einem Monitor arbeitet (Laden, Starten, Eingeben und Speichern eigener Programme)
- Die ersten Befehle und die Adressierungsarten
- Fortgeschrittene Assemblerprogrammierung
- Noch mehr über Schleifen
- Rechnen in Maschinensprache
- Vergleichsbefehle
- Stapeloperationen
- Interruptprogrammierung
- Die verschiedenen Rechner/Programmierhilfsmittel für C16, 116, Plus/4 und C128
- Und viele, viele Beispiele ...

## *UND GESCHRIEBEN HAT DIESES BUCH:*

Said Baloui ist Student und Programmierer professioneller Anwendungen. Er gehört seit langer Zeit zum Team der DATA BECKER Autoren.

***ISBN 3-89011-182-3***