# Programmieren in Maschinensprache, Teil 1

**Programmieren in Maschinensprache, Teil 1**

Die Sprache BASIC lässt die meisten Programmierer vergessen, daß in dem Rechner Ausdrücke wie IF, THEN usw. eine Folge von Bitmuster sind, welche die CPU als ein ausführbares Maschinenprogramm liest. Sobald man aber die Sprache BASIC verlässt, muß man selbst auf diese Ebene des Programmierens in Maschinencode heruntersteigen. Dies ist vor allem dann notwendig, wenn der Rechner zu Steuerungsaufgaben in Verbindung mit der Außenwelt eingesetzt wird. Diese Einführung ist also für alle diejenigen gedacht, die sich bisher nur mit BASIC befasst haben, und die nun etwas tiefer in die Programmierung einsteigen wollen. Daß hierbei wiederum der Prozessor 6502 mit seinen Befehlsvorrat im Vordergrund steht, liegt einfach daran, daß die meisten, auch die neuesten, Home-Computer diese CPU verwenden. Es ist aber gleichgültig,mit welchen Prozessor man das Programmieren in Maschinensprache lernt. Das Umsteigen auf einen anderen Prozessor bedeutet nur die Verwendung eines anderen Befehlsvorrates, das "Denken in Maschinensprache" bleibt das gleiche. In diesen ersten Teil sollen nun zuerst einige Grundbegriffe erläutert werden.

Der Einstieg in die Maschinensprache erfolgt über den MONITOR. Dies ist das Betriebssystem, das nach dem Einschalten des Rechners aktiv wird und von sich aus das weitere Einlesen und Ausführen von Programmen übernimmt. Dieser Monitor ist für das Programmieren in Maschinencode sehr wichtig. Einmal erkennt er Monitorbefehle wie z. B. Ausgeben des Speicherinhaltes auf den Bildschirm oder Starten des Programms. Andererseits enthält er aber Unterprogramme, die in den eigenen Programmen verwendet werden können. Die am häufigsten gebrauchten Unterprogramme sind die Ausgabe eines Zeichen auf ein Ausgabemedium und die Eingabe eines Zeichens in den Rechner.

Der Einstieg in den Monitor erfolgt zum Beispiel beim APPLE II mit CALL–151 aus BASIC, beim Ohio C1P durch Eingabe von M nach dem Einschalten und der AIM 65 ist nach dem Einschalten automatisch im Monitor.

Ein Maschinenprogramm ist eng mit dem Speicher des Computers verknüpft. Jeder Befehl ist an einer festen Adresse gespeichert. Die Adresse ist die Hausnummer jedes Speicherplatzes im Rechner. Wichtig für die Programmausführung ist die Startadresse des Programms. Dies ist die Adresse des Speicherplatzes, in welcher der zuerst auszuführende Befehl gespeichert ist. Diese Adresse wird durch einen Monitorstart-Befehl in den Befehlsfolgezähler übernommen, der von sich aus die Weiterführung des Programms veranlasst.

Wie ist nun solch ein Befehl aufgebaut. Er belegt im Speicher ein, zwei oder drei Byte.
Ein Byte sind 8-bit und bilden den Inhalt eines Speicherplatzes bei einem 8-bit Prozessor.

Das erste Byte enthält den Operationscode. Betrachten wir hierzu Tabelle 1. Hier sind alle Bitmuster, die bei einem 6502 einen Befehl darstellen zusammengestellt. In der linken Spalte sind für diese Bitmuster leicht merkbare Ausdrücke eingegeben. Diese Schreibweise bezeichnet man auch als Assemblerschreibweise.

Auf das Byte mit dem Operationscode können noch ein oder zwei Bytes folgen. Diese enthalten die Adresse des Speicherplatzes auf den die Operation ausgeführt werden soll. Die Angabe dieser Adresse kann auf verschiedene Weisen, den Adressierungsarten erfolgen. Auf diese werden wir in den einzelnen Programmbeispielen eingehen.

**Beispiele für Befehle**

1. Lade den Akkumulator mit den Inhalt der Speicherzelle $1000 ($ bedeutet: Folgende Zahl ist eine Hexadezimalzahl).
   Assemblerschreibweise: LDA $1000
   Darstellung als Bitmuster: AD 00 10

Dies ist also ein 3-Bytebefehl. Gemäß der 6502-Konvention folgt auf den Operationscode erst der niederwertige, dann der höherwertige Adressteil.

2. Vergleiche den Akkumulator mit den Inhalt der folgenden Speicherzelle.
   Assemblerschreibweise: CMP #$7F
   Darstellung als Bitmuster: C9 7F
   Dies ist ein 2-Bytebefehl. Das # Zeichen bedeutet unmittelbare Adressierung. Die Operation bezieht sich auf den Inhalt der auf den Operationscode folgenden Speicherzelle.

3. Schiebe den Inhalt des Akkumulators eine Stelle links
   Assemblerschreibweise: ASL
   Darstellung als Bitmuster: OA
   Dies ist ein 1-Bytebefehl, denn die Angabe einer Adresse ist hier nicht notwendig.

In der nächsten Folge werden wir uns mit der logischen Struktur der CPU 6502, dem Aufstellen von Programmen und der "Assemblierung von Hand" befassen:

Stichpunkte zum Teil 1:

- MONITOR
- ADRESSE
- BEFEHLSFOLGEZÄHLER
- BEFEHL
- 1, 2 und 3 BYTEBEFEHLE

E. Flögel

| Befehle | symb. Code | Wirkung | ADRESSIERUNGSARTEN UNM | ABS | ABS,X | ABS,Y | S0 | S0,X | S0,Y | (IND,X) | (IND),Y | REL | IND | ACCU | IMPL | N | Z | C | I | D | V |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Transport | LDA | M → A | A9 | AD | BD | B9 | A5 | B5 | | A1 | B1 | | | | | X | X | – | – | – | – |
| | LDX | M → X | A2 | AE | | BE | A6 | | B6 | | | | | | | X | X | – | – | – | – |
| | LDY | M → Y | A0 | AC | BC | | A4 | B4 | | | | | | | | X | X | – | – | – | – |
| | STA | A → M | | 8D | 9D | 99 | 85 | 95 | | 81 | 91 | | | | | – | – | – | – | – | – |
| | STX | X → M | | 8E | | | 86 | | 96 | | | | | | | – | – | – | – | – | – |
| | STY | Y → M | | 8C | | | 84 | 94 | | | | | | | | – | – | – | – | – | – |
| | TAX | A → X | | | | | | | | | | | | | AA | X | X | – | – | – | – |
| | TAY | A → Y | | | | | | | | | | | | | A8 | X | X | – | – | – | – |
| | TXA | X → A | | | | | | | | | | | | | 8A | X | X | – | – | – | – |
| | TYA | Y → A | | | | | | | | | | | | | 98 | X | X | – | – | – | – |
| | TXS | X → S | | | | | | | | | | | | | 9A | – | – | – | – | – | – |
| | TSX | S → X | | | | | | | | | | | | | BA | X | X | – | – | – | – |
| | PLA | S+1 → S, Ms → A | | | | | | | | | | | | | 68 | X | X | – | – | – | – |
| | PHA | A → Ms, S−1 → S | | | | | | | | | | | | | 48 | – | – | – | – | – | – |
| | PLP | S+1 → S, Ms → P | | | | | | | | | | | | | 28 | | | | | | |
| | PHP | P → Ms, S−1 → S | | | | | | | | | | | | | 08 | – | – | – | – | – | – |
| Arithmetische | ADC | A+M+C → A | 69 | 6D | 7D | 79 | 65 | 75 | | 61 | 71 | | | | | X | X | X | – | – | X |
| | SBC | A−M−$\overline{C}$ → A | E9 | ED | FD | F9 | E5 | F5 | | E1 | F1 | | | | | X | X | X | – | – | X |
| | INC | M+1 → M | | EE | FE | | E6 | F6 | | | | | | | | X | X | – | – | – | – |
| | DEC | M−1 → M | | CE | DE | | C6 | D6 | | | | | | | | X | X | – | – | – | – |
| | INX | X+1 → X | | | | | | | | | | | | | E8 | X | X | – | – | – | – |
| | DEX | X−1 → X | | | | | | | | | | | | | CA | X | X | – | – | – | – |
| | INY | Y+1 → Y | | | | | | | | | | | | | C8 | X | X | – | – | – | – |
| | DEY | Y−1 → Y | | | | | | | | | | | | | 88 | X | X | – | – | – | – |
| Logische | AND | A ∧ M → A | 29 | 2D | 3D | 39 | 25 | 35 | | 21 | 31 | | | | | X | X | – | – | – | – |
| | ORA | A ∨ M → A | 09 | 0D | 1D | 19 | 05 | 15 | | 01 | 11 | | | | | X | X | – | – | – | – |
| | EOR | A ⊻ M → A | 49 | 4D | 5D | 59 | 45 | 55 | | 41 | 51 | | | | | X | X | – | – | – | – |
| Vergleichs- | CMP | A−M | C9 | CD | DD | D9 | C5 | D5 | | C1 | D1 | | | | | X | X | X | – | – | – |
| | CPX | X−M | E0 | EC | | | E4 | | | | | | | | | X | X | X | – | – | – |
| | CPY | Y−M | C0 | CC | | | C4 | | | | | | | | | X | X | X | – | – | – |
| | BIT | A ∧ M | | 2C | | | 24 | | | | | | | | | 7 | X | – | – | – | 6 |
| Verzweigungs- | BCC | BRANCH ON C=0 | | | | | | | | | 90 | | | | | – | – | – | – | – | – |
| | BCS | BRANCH ON C=1 | | | | | | | | | B0 | | | | | – | – | – | – | – | – |
| | BEQ | BRANCH ON Z=1 | | | | | | | | | F0 | | | | | – | – | – | – | – | – |
| | BNE | BRANCH ON Z=0 | | | | | | | | | D0 | | | | | – | – | – | – | – | – |
| | BMI | BRANCH ON N=1 | | | | | | | | | 30 | | | | | – | – | – | – | – | – |
| | BPL | BRANCH ON N=0 | | | | | | | | | 10 | | | | | – | – | – | – | – | – |
| | BVC | BRANCH ON V=0 | | | | | | | | | 50 | | | | | – | – | – | – | – | – |
| | BVS | BRANCH ON V=1 | | | | | | | | | 70 | | | | | – | – | – | – | – | – |
| | JMP | | | 4C | | | | | | | | | | 6C | | – | – | – | – | – | – |
| | JSR | | | 20 | | | | | | | | | | | | – | – | – | – | – | – |
| Schiebe- | ASL | | | 0E | 1E | | 06 | 16 | | | | | | | 0A | X | X | X | – | – | – |
| | LSR | | | 4E | 5E | | 46 | 56 | | | | | | | 4A | 0 | X | X | – | – | – |
| | ROL | | | 2E | 3E | | 26 | 36 | | | | | | | 2A | X | X | X | – | – | – |
| | ROR | | | 6E | 7E | | 66 | 76 | | | | | | | 6A | X | X | X | – | – | – |
| Status-Register | CLC | C=0 | | | | | | | | | | | | | 18 | – | – | 0 | – | – | – |
| | CLD | D=0 | | | | | | | | | | | | | D8 | – | – | – | – | 0 | – |
| | CLI | I=0 | | | | | | | | | | | | | 58 | – | – | – | 0 | – | – |
| | CLV | V=0 | | | | | | | | | | | | | B8 | – | – | – | – | – | 0 |
| | SEC | C=1 | | | | | | | | | | | | | 38 | – | – | 1 | – | – | – |
| | SED | D=1 | | | | | | | | | | | | | F8 | – | – | – | – | 1 | – |
| | SEI | I=1 | | | | | | | | | | | | | 78 | – | – | – | 1 | – | – |
| Versch. | NOP | NO OPER | | | | | | | | | | | | | EA | – | – | – | – | – | – |
| | RTS | RETURN F. SUB | | | | | | | | | | | | | 60 | – | – | – | – | – | – |
| | RTI | RETURN F. INT | | | | | | | | | | | | | 40 | | | | | | |
| | BRK | BREAK | | | | | | | | | | | | | 00 | – | – | – | 1 | – | – |

Tabelle 1

# Programmieren in Maschinensprache, Teil 2

### 2.1 Programmiermodell der CPU 6502

Einen Überblick über die Befehle, die ein Mikroprozessor ausführen kann, erhält man durch das Programmiermodell des Hardware-Bausteins. Für die 6502-CPU ist dies in Abbildung 2.1 dargestellt. Es gibt vier 8-Bit Register, den Akkumulator, das X- und Y-Register und das Statusregister. Der Programm- oder auch Befehlsfolgezähler hat 16 Bit und damit einen Adressenumfang von 0 bis 65535.

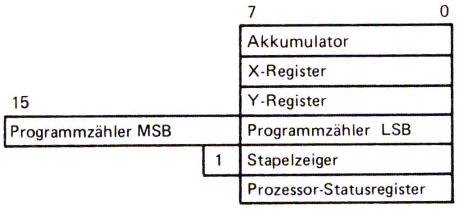



**Abbildung 2.1**
**Programmiermodell 6502**

Als letztes bleibt noch der Stapelzeiger. Dieser zeigt auf einen besonderen Speicherbereich, auf die Adressen $ 100 – $ 1FF, dem Stapel. Zur Adressierung benutzt er nur 8 Bit, das 9. Bit ist immer 1 und wird automatisch vom Prozessor hinzugefügt.

Welche Aufgaben haben nun die einzelnen Register?
Das zentrale Register ist der Akkumulator. Alle Rechenoperationen werden über den Akkumulator ausgeführt. Wird zum Akkumulatorinhalt der Inhalt einer Speicherzelle hinzuaddiert, so ist das Ergebnis dieser Operation der neue Inhalt des Akkumulators. Auch das Umspeichern des Inhalts einer Speicherzelle in eine andere Speicherzelle erfolgt über den Akkumulator. Dieses Umspeichern kann aber auch über die Indexregister geschehen. Weiter können diese Register als Zählregister verwendet werden. Durch einen Befehl INX z.B. wird der Inhalt des X-Registers um Eins erhöht, durch DEY der Inhalt des Y-Registers um Eins erniedrigt. Ferner können mit ihnen Adressen verändert, indiziert werden. Daher auch der Name Indexregister.
Von dieser Möglichkeit werden wir bei späteren Programmen häufig Gebrauch machen.

Das Statusregister zeigt den augenblicklichen Zustand eines Programmes an. In den einzelnen Bits wird das Ergebnis einer Operation festgehalten (Abbildung 2.2)

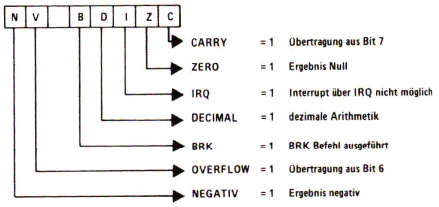



**Abbildung 2.2**
**Belegung der Bit im Statusregister**

Das Zero-Bit wird z.B. Eins, wenn der Akkumulatorinhalt gleich Null wird. Das Carry-Bit wird gesetzt, wenn bei einer Addition ein Übertrag in die nächste Stelle auftritt. In der rechten Spalte von Tabelle 1 aus dem letzten Heft wird gezeigt, welche Operationen die einzelnen Bits im Statusregister verändern können, dabei zeigt ein X eine mögliche Änderung an. Ein LDA-Befehl beeinflußt also nur das N und Z-Bit, alle anderen nicht, während ein STA-Befehl kein Bit im Statusregister verändert.
Der Stapelzeiger zeigt, wie sein Name schon besagt, auf einen freien Speicherplatz im Stapel. Durch einen 1-Byte-Befehl PHA (Push Accumulator) wird der Inhalt des Akkumulators dort hingeschrieben, und der Stapelzeiger automatisch auf die nächste Speicherzelle gesetzt. Bei einem PLA (Pull Accumulator) wird der Stapelzeiger zuerst zurückgesetzt und der Inhalt dieser Zelle in den Akkumulator übernommen. Dabei ist zu beachten, daß die oberste Zelle des Stapels die Adresse $ 1FF ist und der Stapel zur Adresse $ 100 hin aufgebaut wird. Dieser Stapelspeicher hat noch eine weitere wichtige Aufgabe. Er übernimmt bei einem Sprung in ein Unterprogramm automatisch die augenblickliche Adresse des Programmzählers. Von dort wird sie beim Rücksprung wieder in den Programmzähler übernommen.

Der Befehlsfolgezähler enthält also immer die Adresse des nächsten ausführbaren Befehls. Er wird nur durch die Sprungbefehle (JMP, JSR) verändert.

In Abbildung 2.3 sind noch einmal alle Transportbefehle für die Datenübertragung zwischen den Registern und dem Speicher gezeigt. Man sieht, daß beim 6502 kein Befehl existiert, der den Datentransfer zwischen Speicherplätzen direkt ausführt, und ein direkter Austausch des Inhalts von X- und Y-Registern ist auch nicht möglich.

Wenn man, nach der Kenntnis einer Maschinensprache, auf die Sprache eines anderen Prozessors umsteigt, so sollte man sich dessen logische Struktur genau ansehen. Daraus kann man schon im voraus feststellen, welchen Befehlsvorrat er umfaßt, und welche Wirkung die einzelnen Befehle haben.

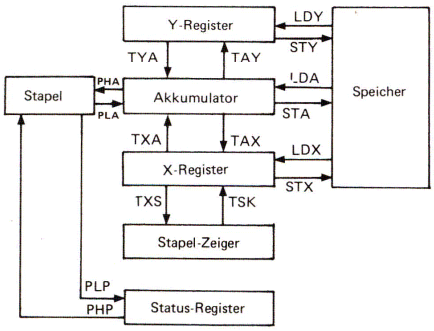



**Abbildung 2.3**
**Datenübertragung zwischen den einzelnen Registern und dem Speicher**

## 2.2 Ein erstes Beispiel und die Papier- und Bleistiftmethode.

Wenn man in einer höheren Programmiersprache, wie z. B. in BASIC zwei Zahlen addiert, so ist das Programm leicht hinzuschreiben.

```
10   A = 5
20   B = 3
30   C = A + B
40   PRINT C
50   END
```

Will man das gleiche in Maschinensprache machen, so sind vor dem Programmschreiben weitere Überlegungen notwendig.

Folgende Fragen müssen zuerst beantwortet werden:
Wo sind die Zahlen gespeichert?
Sind es Festkommazahlen oder Gleitkommazahlen?
Wo soll das Programm beginnen?
Gibt es im Monitor ein Programm, das den Inhalt einer Speicherzelle ausdruckt?

Die Beantwortung dieser Fragen wollen wir bei der Umsetzung dieses BASIC-Programms in ein Maschinenprogramm vornehmen. Wir beginnen mit einer Formulierung in der Assemblerschreibweise. Die Befehle lauten der Reihenfolge nach:

| | |
|---|---|
| LDA #$05 | Lade den Akkumulator mit 05. Es wird die unmittelbare Adressierung verwendet. Die Zahl 05 ist nach dem Operationscode gespeichert und ist eine Festkommazahl. |
| CLC | Lösche das Carrybit für den nachfolgenden Befehl: Addiere mit Übertrag (Carry) |
| ADC #$03 | Addiere mit Übertrag eine 03. Es wird wieder die unmittelbare Adressierung verwendet. Das Ergebnis ist automatisch im Akkumulator. |
| JSR PRTBYT | – PRTBYT ist ein Monitorunterprogramm, das den Inhalt des Akkumulators als zwei Hexadezimalzahlen ausgibt. |
| BRK | Wenn die Ausgabe beendet wird, breche hier das Programm ab. |

Das Programm lautet also:

```
LDA # $05
CLC
ADC # $03
JSR PRTBYT
BRK
```

Einen Überblick über dezimale und hexadezimale Adressen und Speichergröße gibt Abbildung 2.4. Links sind die Adressen dezimal, rechts hexadezimal angegeben. Der Adressbereich von 0 bis $ 400 umfaßt 1K Byte Speicherplatz, der Adressbereich $1000 – $2000 4K Byte.

Dieses Programm wollen wir nun mit der "Papier- und Bleistift"-Methode in die Bitmuster eines ausführbaren Maschinenprogramms umsetzen. Dies ist zwar die unterste Methode der Assemblierung, aber dabei können weitere Kenntnisse der Programmierung erworben werden. Zuerst muß aber die Frage beantwortet werden: Wo beginnt das Programm?
Grundsätzlich kann das Programm überall dort beginnen, wo Speicherplatz im Rechner vorhanden ist. Zwei Bereiche sollte man aber schon von vornherein nicht benutzen. Das ist einmal die "Seite NULL" oder Zero Page, die, wie wir gleich sehen werden, sehr nützlich für

Hilfszellen ist und zum anderen der Stapelspeicher, da ihn auch der Prozessor benötigt. Damit sind die Speicherplätze 0 bis $1FF nicht verwendbar.

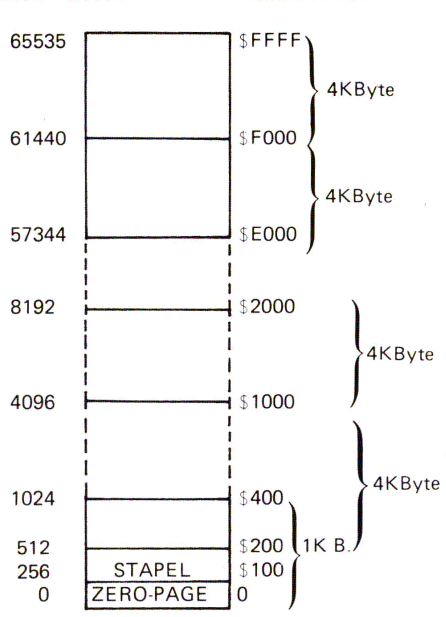



**Abbildung 2.4**
**Dezimale und hexadezimale Adressen eines 64K Byte Speichers**

Wir wollen annehmen, daß ab Adresse $800 freier Platz für unser Programm ist. Damit kann der erste Befehl umgesetzt werden. In der Tabelle 1 (ELCOMP Juni-Heft, S. 59) finden wir für LDA unmittelbar das Bitmuster A9 und damit die erste Zeile:

```
$800    A9 05   LDA # $05
```

A9 ist der Operationscode, 05 die Zahl, die unmittelbar darauf folgt. Dies sind 2 Byte und die nächste Zeile beginnt in $802.

```
$802    18      CLC
```

18 ist das Bitmuster, welches das Übertragungsbit löscht. Wir finden es in der Tabelle 1 bei den Status-Register-Befehlen. Es folgt die Zeile mit der Addition ADC (Add with Carry).
Bei der Addition wird das Übertragungsbit zum Ergebnis addiert. Deshalb wurde es in der vorangegangenen Zeile gelöscht.

```
$803    69 03   ADC # $03
```

Wieder wurde die unmittelbare Adressierung verwendet. Das Bitmuster 69 für ADC unmittelbar (#) findet man bei den arithmetischen Befehlen.

Nun folgt der Aufruf des Unterprogramms PRTBYT, das den Inhalt des Akkumulators auf den Bildschirm ausgibt.

Beim APPLE II Computer beginnt dieses Unterprogramm bei der Adresse $FDED. Die Ausgabezeile für diesen Rechner lautet dann, mit 20 als Bitmuster für den Befehl JSR (Jump Subroutine):

```
$805    20 ED FD   JSR PRTBYT
```

Bei 6502 Prozessoren folgt bei Adressangaben auf das Byte mit dem Operationscode das niederwertige Adressbyte LSB (Least Significant Byte), danach das höherwertige Adressbyte (Most Significant Byte). Danach wird das Programm mit

```
$808    00      BRK
```

abgebrochen. Dabei findet bei den meisten Rechnern ein Rücksprung in den Monitor statt.
Das Programm lautet somit:

```
$800    A9 05   LDA #$05
$802    18      CLC
$803    69 03   ADC # $03
$805    20 ED FD JSR PRTBYT
$808    00      BRK
```

Die Speicherzellen von $800 bis $808 haben damit folgenden Inhalt:

```
$800 : A9 05 18 69 03 20 ED FD
$808 : 00
```

Es soll im Augenblick nicht darauf eingegangen werden, wie man diese Bitmuster in den Rechner schreibt, und wie man das Programm startet. Es soll vielmehr gezeigt werden, wie sich das Programm ändert, wenn man andere Adressierungsarten verwendet.

Die Programmieraufgabe bleibt die gleiche, die beiden Zahlen 5 und 3 sind aber in 2 Zellen der Zero-Page (S0) gespeichert.
Die Zahl 5 in Zelle $10 und die 3 in Zelle $11. Damit erhält man:

```
$800    A5 10   LDA $10 ;Hole den Inhalt
                        der Zelle $10 in
```

```
                              d. Akkumulator
$802   18          CLC        ; Carrybit löschen
$803   65 11       ADC $11    ; Addiere dazu d.
                                Inhalt d.Zelle$11.
$805   20 ED FD    JSR PRTBYT  ;Ausgabe
$808   00          BRK        ; Abbruch
```

A5 ist das Bitmuster für LDA mit dem Inhalt einer Zelle aus der Zero Page (S0), 65 das Bitmuster für ADC mit dem Inhalt einer Zelle in der Zero Page.

Beim letzten Beispiel wollen wir annehmen, daß die beiden Zahlen irgendwo im Rechner z.B. in den Zellen $200A und $3005 gespeichert sind.

Das Programm hat dann folgendes Aussehen:

```
$800   AD 0A 20   LDA $200A  ; Hole den In-
                              halt der Zelle
                              $200A
$803   18         CLC        ;Carry-Bit lö-
                              schen
$804   6D 05 30   ADC $3005  ;Addiere dazu
                              den Inhalt v.
                              $3005
$807   20 ED FD   JSR PRTBYT;Ausgabe
$809   00         BRK        ;Abbruch
```

Hier ist AD das Bitmuster für den Befehl LDA mit dem Inhalt einer absoluten Adresse und 6D das Bitmuster für ADC mit dem Inhalt einer absoluten Adresse.
Das letzte Programm belegt 2 Byte mehr Speicherplatz. Benötigt man in einem Programm viele Hilfszellen, so legt man diese in die Zero-Page. Die Programme werden dadurch kürzer.

In der nächsten Folge werden wir Programme mit Verzweigungen betrachten und dabei auf die Darstellung von positiven und negativen Zahlen eingehen. Ferner sollen weitere Adressierungsarten besprochen werden.

Stichpunkte zu Teil 2:

- PROGRAMMIERMODELL 6502
- CPU-REGISTER
- UNMITTELBARE ADRESSIERUNG
- ADRESSIERUNG DER ZERO PAGE
- ABSOLUTE ADRESSIERUNG

**E. Flögel** ■

# Programmieren in Maschinensprache, Teil 3

**Programmieren in Maschinensprache mit 6502**
**Teil 3**

Im Teil 2 des Kurses haben wir ein Programm in Maschinencode übersetzt, das keine Verzweigungen enthielt, sondern von Anfang bis Ende geradlinig durchlaufen wurde. Im Folgenden werden wir nun Programme mit Verzweigungen betrachten.

### 3.1 Programmverzweigungen

Die meisten Programme werden Programmschleifen enthalten, die solange durchlaufen werden, bis eine Bedingung erfüllt ist. Dann wird diese Schleife verlassen und das Programm fortgesetzt.

Bedingungen sind z. B., ob der Inhalt eines Registers oder einer Speicherzelle Null ist, oder ob eine Zahl in einem Register größer, gleich oder kleiner als eine Zahl in einer Speicherzelle ist. Durch Vergleiche, aber auch durch Operationen werden die Bits im Statusregister gesetzt (s. Abb. 2.2). Die Verzweigungsbefehle überprüfen diese Bits und führen danach eine Verzweigung aus oder auch nicht. Das einfachste Beispiel ist eine Zeitschleife:

Der Inhalt des X-Registers wird solange um Eins erniedrigt, bis der Inhalt des Registers Null ist.

Programm:

```
    LDX   # $A0    ; Lade das X-Reg. A0
M   DEX            ; Erniedrige X-Reg. um
                   ; Eins
    BNE   M        ; Springe, wenn nicht
                   ; Null nach M zurück
    BRK            ; sonst breche Programm
                   ; ab
```

Dieses Programm wird nun von Hand übersetzt. Anfangsadresse ist $800.

```
800    A2 A0    LDX # $A0
802    CA    M  DEX
803    D0 —     BNE M
805    00       BRK
```

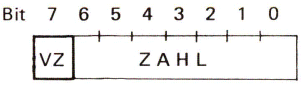
Noch nicht angegeben ist in dem 2-Byte-Befehl die Zahl der Byte, die zurückgesprungen werden. Dazu sind 2 weitere Überlegungen notwendig. Die Verzweigungsbefehle benutzen die relative Adressierung. Das bedeutet, der Befehlsfolgezähler wird um die angegebene Zahl von Bytes erhöht oder erniedrigt und das Programm an dieser Stelle fortgesetzt.

Welchen Inhalt hat nun der Befehlsfolgezähler?

Bei den 6502 Systemen zeigt er auf die Adresse des nächsten Befehls, in unserem Fall auf den BRK in Zelle 805. Zur Zelle 802 muß nun um 3 Byte zurückgesprungen werden. In Zelle 804 muß also die Hexadezimalzahl —3 stehen. Um diese zu bestimmen, müssen wir also negative Zahlen einführen.

### 3.2 Positive und negative Zahlen

Die Kennzeichnung von positiven und negativen Zahlen erfolgt durch Bit 7 der Zahl. Ist dieses Bit = 1,




so ist es eine negative Zahl, ist es = 0, so ist es eine positive Zahl.

Damit erhält man für positive Zahlen:

0 = $00 = % 0000 0000
1 = $01 = % 0000 0001
2 = $02 = % 0000 0010
.
.
127 = $7F = % 0111 1111

Negative Zahlen werden dagegen im 2-er Complement dargestellt. Complementieren einer Zahl bedeutet das Vertauschen von 0 und 1 im Bitmuster. Bei der Bildung des 2-Complements wird zu diesem neuen Bitmuster noch eine Eins hinzuaddiert.

Zur Bildung der Zahl —1 benötigen wir also das Bitmuster für die Zahl +1 = % 0000 0001.

Dieses Bitmuster complementiert, ergibt % 1111 1110. Dazu 1 addiert ergibt:

```
        % 1111 1110
+       % 0000 0001
-----------------------
        % 1111 1111 = $FF
```

Berechnen wir auf diese Weise die Zahl –3.

```
    + 3 =   % 0000 0011
Complement  % 1111 1100
+ 1         + % 0000 0001
            --------------------
            % 1111 1101 = $FD
```

Für negative Zahlen erhält man:

```
– 1 = $FF = % 1111 1111
– 2 = $FE = % 1111 1110
– 3 = $F0 = % 1111 1101
      .
      .
      .
– 128= $80 = % 1000 0000
```

Der Zahlenbereich für eine vorzeichenbehaftete 8-Bit-Zahl reicht also von –128 bis + 127. Dies sind dann auch die Grenzen bei der relativen Adressierung der Verzweigungsbefehle.
Unser Programm einer Zeitschleife lautet damit:

```
800    A2 A0    LDX # $A0
802    CA     M DEX
803    D0 FD    BNE M
805    00       BRK
```

Für die Berechnung der Verzweigungen benutzt man am besten die Tabellen in Abbildung 3.1 und 3.2.

| LSD / MSD | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 2 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 3 | 48 | 49 | 50 | 51 | 52 | 53 | 54 | 55 | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |
| 4 | 64 | 65 | 66 | 67 | 68 | 69 | 70 | 71 | 72 | 73 | 74 | 75 | 76 | 77 | 78 | 79 |
| 5 | 80 | 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 | 91 | 92 | 93 | 94 | 95 |
| 6 | 96 | 97 | 98 | 99 | 100 | 101 | 102 | 103 | 104 | 105 | 106 | 107 | 108 | 109 | 110 | 111 |
| 7 | 112 | 113 | 114 | 115 | 116 | 117 | 118 | 119 | 120 | 121 | 122 | 123 | 124 | 125 | 126 | 127 |

**Abbildung 3.1 Vorwärtsverzweigungen**

| LSD / MSD | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 128 | 127 | 126 | 125 | 124 | 123 | 122 | 121 | 120 | 119 | 118 | 117 | 116 | 115 | 114 | 113 |
| 9 | 112 | 111 | 110 | 109 | 108 | 107 | 106 | 105 | 104 | 103 | 102 | 101 | 100 | 99 | 98 | 97 |
| A | 96 | 95 | 94 | 93 | 92 | 91 | 90 | 89 | 88 | 87 | 86 | 85 | 84 | 83 | 82 | 81 |
| B | 80 | 79 | 78 | 77 | 76 | 75 | 74 | 73 | 72 | 71 | 70 | 69 | 68 | 67 | 66 | 65 |
| C | 64 | 63 | 62 | 61 | 60 | 59 | 58 | 57 | 56 | 55 | 54 | 53 | 52 | 51 | 50 | 49 |
| D | 48 | 47 | 46 | 45 | 44 | 43 | 42 | 41 | 40 | 39 | 38 | 37 | 36 | 35 | 34 | 33 |
| E | 32 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 |
| F | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 |

**Abbildung 3.2 Rückwärtsverzweigungen**

Die Berechnung der Sprungweite ist bei der Assemblierung von Hand eine der häufigsten Fehlerquellen.

### 3.3 Vergleiche

Ein Vergleich findet immer zwischen einem Register (Akkumulator, X- oder Y-Register) und einer Speicherzelle statt. Durch den Vergleich werden die Bit N (Negativ) Z (Zero) und C (Carry) gesetzt. Dies ist in Abbildung 3.3 dargestellt.

| Vergleich | N | Z | C |
|---|---|---|---|
| A, X, Y < M | 1* | 0 | 0 |
| A, X, Y = M | 0 | 1 | 1 |
| A, X, Y > M | 0* | 0 | 1 |

*Vergleich im 2-er Complement

**Abbildung 3.3**
**Setzen der Bits im Statusregister durch Vergleichsbefehle**

Ist der Inhalt des Akkumulators (X-, Y-Register) kleiner als der Inhalt einer Speicherzelle, so wird das Zero- und das Carry-Bit im Statusregister auf Null gesetzt.

Für die beiden Bits werden die Zahlen als Weite zwischen 0 und 255 betrachtet. Das Setzen des N-Bits erfolgt als Vergleich im 2-er Complement, also für einen Wertebereich von –128 bis + 127.
Dazu ein Beispiel: Der Inhalt des Akkumulators ist SFD, der Inhalt einer Speicherzelle ist $ 00. Bei einem Vergleich ist A > M (252 > 00) somit wird C = 1 und Z = 0.
Für die verschiedenen Möglichkeiten einer Verzweigung erhält man:

| Verzweigung nach MARKE, wenn | mit |
|---|---|
| A < M | BCC MARKE |
| A ≤ M | BCC MARKE |
| | BEQ MARKE |
| A = M | BEQ MARKE |
| A ≥ M | BCS MARKE |
| A > M | BEQ NICHT-MARKE |
| | BCS MARKE |

Als einfaches Beispiel für Vergleiche und Verzweigungen soll folgendes Programm betrachtet werden:
Über das Tastenfeld soll ein Zeichen eingegeben werden. Dieses Zeichen wird daraufhin untersucht, ob es ein Hexadezimalzei-

chen, also 0 bis 9 und A bis F ist. Ist dies der Fall, so wird diese Zahl in einer Zelle EIN mit der Adresse $10 gespeichert. Wenn das Zeichen keiner Hexadezimalzahl entspricht, wird das Programm mit $00 in EIN verlassen.

Für die Eingabe verwenden wir ein Unterprogramm GETCHR, das in den meisten Monitoren enthalten ist. Das Unterprogramm frägt laufend das Tastenfeld ab, ob eine Taste gedrückt wird. Nach einem Tastendruck kehrt dieses Unterprogramm mit dem ASCII-Zeichen im Akkumulator in das Hauptprogramm zurück.

In Abbildung 3.4 sind alle ASCII-Zeichen zusammengestellt.

| LSD \ MSD | | 0 000 | 1 001 | 2 010 | 3 011 | 4 100 | 5 101 | 6 110 | 7 111 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0000 | NUL | DLE | SP | 0 | @ | P | ` | p |
| 1 | 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | 0010 | STX | DC2 | " | 2 | B | R | b | r |
| 3 | 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 4 | 0100 | EOT | DC4 | $ | 4 | D | T | d | t |
| 5 | 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | 0111 | BEL | ETB | ' | 7 | G | W | g | w |
| 8 | 1000 | BS | CAN | ( | 8 | H | X | h | x |
| 9 | 1001 | HT | EM | ) | 9 | I | Y | i | y |
| A | 1010 | LF | SUB | * | : | J | Z | j | z |
| B | 1011 | VT | ESC | + | ; | K | [ | k | { |
| C | 1100 | FF | FS | , | < | L | \ | l | \| |
| D | 1101 | CR | GS | - | = | M | ] | m | } |
| E | 1110 | SO | RS | . | > | N | ↑ | n | ~ |
| F | 1111 | SI | VS | / | ? | O | ← | o | DEL |

**ASCII-Zeichen**

```
0800 A900      7          LDA #$00
0802 8510      8          STA $10
0804 200CFD    9          JSR GETCHR
0807           10 ;
0807           11 ; NUR FUER APPLE
0807 297F      12         AND #%01111111
0809           13 ;
0809 C930      14         CMP #$30
080B 9013      15         BCC ENDE
080D C947      16         CMP #$47
080F B00F      17         BCS ENDE
0811 C93A      18         CMP #$3A
0813 9007      19         BCC ZAHL
0815 C941      20         CMP #$41
0817 9007      21         BCC ENDE
0819 18        22         CLC
081A 6909      23         ADC #$09
081C 290F      24 ZAHL    AND #%00001111
081E 8510      25         STA $10
0820 00        26 ENDE    BRK
```

**Abbildung 3.5**

Dieses Programm wurde für den APPLE II übersetzt. Abbildung 3.5. Hier mußte noch eine Zeile mit AND #% 0111 1111 nach dem Unterprogrammsprung GETCHR eingefügt werden, da Bit 7 beim APPLE immer 1 ist.

Als Übung kann man dieses Programm einmal von Hand übersetzen und dabei die Sprungweiten bestimmen und überprüfen. Im Teil 4 werden wir uns mit der Verwendung von Unterprogrammen beschäftigen.

Stichpunkte zu Teil 3:

- Programmverzweigungen
- Positive und negative Zahlen
- Relative Adressierung
- Vergleiche

E. Flögel ■

# Der Challenger

**Weltneuheit**

Im Bild sehen Sie den neuen Supercomputer von OHIO Scientific, den wir im Juni-Heft 1981 schon kurz beschrieben haben (Seite 8).

Der Challenger V besitzt zwei Minifloppylaufwerke und zwei 5inch Winchester-Festplatten 6502 CPU plus (Z-80 oder 68000 oder Z8000) 132 Zeichen/Zeile, 200 KRAM usw.

Leider hört man in Europa zur Zeit wenig über OHIO Scientific.

# Programmieren in Maschinensprache, Teil 4

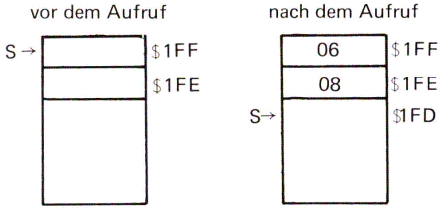
**Programmieren in Maschinensprache, Teil 4**
In dieser Folge wollen wir uns mit der Verwendung von Unterprogrammen beschäftigen. Unterprogramme sind selbständige Programmteile, die durch einen Unterprogrammaufruf JSR (Jump Subroutine) gestartet werden. Die Rückkehr in das Hauptprogramm erfolgt durch den Befehl RTS (Return from Subroutine).

## 4.1 Unterprogrammaufrufe

Als Beispiel wollen wir den Unterprogrammaufruf JSR GETCHR im Programm ASCII HEX aus der letzten Folge betrachten. Die ersten Programmzeilen lauteten dort:

```
0800 A900      7   LDA #$00
0802 8510      8   STA $10
0804 200CFD    9   JSR GETCHR
0807 297F     10   AND #%01111111
0809 C930     11   CMP #$30
```

In der Speicherzelle $804 ist der Unterprogrammaufruf programmiert. Bei der Ausführung dieses Befehles wird die Adresse des nächsten Befehls (um Eins erniedrigt) in den Stapelspeicher übernommen.




Der Stapelspeicher ist bei den 6502-Systemen ein fester Speicherbereich mit dem TOS (Top of Stack) bei $1FF.
Der Stapelzeiger S zeigt immer auf die nächste freie Speicherzelle im Stapel.

```
FD0C-   A4 24     LDY   $24
FD0E-   B1 28     LDA   ($28),Y
FD10-   48        PHA
FD11-   29 3F     AND   #$3F
FD13-   09 40     ORA   #$40
FD15-   91 28     STA   ($28),Y
FD17-   68        PLA
FD18-   6C 38 00  JMP   ($0038)
FD1B-   E6 4E     INC   $4E
FD1D-   D0 02     BNE   $FD21
FD1F-   E6 4F     INC   $4F
FD21-   2C 00 C0  BIT   $C000
FD24-   10 F5     BPL   $FD1B
FD26-   91 28     STA   ($28),Y
FD28-   AD 00 C0  LDA   $C000
FD2B-   2C 10 C0  BIT   $C010
FD2E-   60        RTS
FD2F-   20 0C FD  JSR   $FD0C
FD32-   20 A5 FB  JSR   $FBA5
FD35-   20 0C FD  JSR   $FD0C
*
```

**Abb. 4.2**
**Programm GETCHR**

Wir betrachten nun das Unterprogramm GETCHR des APPLE II Computers, beginnend bei der Adresse $FD0C.

Das Programm wird linear durchlaufen, bis zur Adresse $FD2E. Dort erfolgt der Rücksprung ins Hauptprogramm durch RTS. Die Tastenabfrage geschieht bei Adresse $FD21. Solange Bit 7 der Zelle $C000 Null ist, ist keine Taste gedrückt, und es wird nach $FD1B zurückgesprungen. Dabei wird der Inhalt der beiden Zellen $4E und $4F laufend um Eins erhöht. Dieser Inhalt kann als Zufallszahl verwendet werden. Auf andere Einzelheiten des Programmes wird später noch eingegangen.

Bei der Ausführung des Rücksprungbefehls RTS wird die Adresse vom Stapel in den Befehlsfolgezähler übernommen, um Eins erhöht, und das Programm bei $807 weitergeführt. Der Stapelzeiger wird zurückgesetzt und steht nachher wieder bei $1FF.

Natürlich kann aus einem Unterprogramm wiederum in ein anderes Unterprogramm gesprungen werden. Einen solchen Programmablauf zeigt Abbildung 4.3

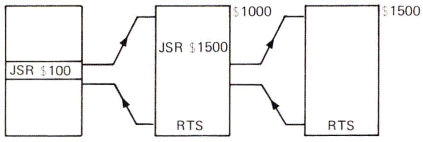



**Abbildung 4.3:**
**Geschachtelte Unterprogrammaufrufe**

Im Stapelspeicher können, wenn sonst keine anderen Daten gespeichert werden, maximal 128 Unterprogrammsprünge abgelegt werden. Dies ist eine Schachtelung, die bei normalen Programmen niemals auftritt.

### 4.2 Retten des Registerinhaltes

In den meisten Fällen werden in einem Unterprogramm die Registerinhalte verändert. Soll aber der Inhalt eines Registers im Hauptprogramm, nach dem Durchlaufen eines Unterprogrammes, weiter verwendet werden, so muß dieser Inhalt gerettet werden.

Dies kann im Hauptprogramm oder im Unterprogramm geschehen. Sind die im Unterprogramm benutzten Register bekannt, so brauchen nur diese im Hauptprogramm zwischengespeichert werden. Die einfachste Methode ist aber, ein im Unterprogramm benutztes Register auch dort zu retten. Der Beginn eines Unterprogramms kann dann folgendermaßen aussehen:

```
UP    PHA    ; Akkumulator in den Stapelspeicher
      TXA    ; X → A
      PHA    ; X-Register in den Stapelspeicher
      TYA    ; Y → A
      PHA    ; Y-Register in den Stapelspeicher
```

Vor einem Rücksprung müssen dann die Register wieder geladen werden. Das Ende dieser Unterprogramme sieht dann so aus:

```
      PLA    ; Y-Register laden
      TAY
      PLA    ; X-Register laden
      TAX
      PLA    ; Akkumulator laden
      RTS    ; Rücksprung
```

Eine andere Möglichkeit besteht darin, die Register nicht in Stapel, sondern in Hilfszellen zu speichern.

### 4.3 Übergabe von Daten an ein Unterprogramm

Auf die folgenden 3 Arten können Daten aus einem Hauptprogramm an ein Unterprogramm oder von einem Unterprogramm zurück an das Hauptprogramm übergeben werden.

1. Daten werden über die Register ausgetauscht. Bei den meisten Eingabeprogrammen, bei denen ein Zeichen über das Tastenfeld eingegeben wird, ist dieses nach dem Verlassen des Unterprogramms im Akkumulator.

2. Die Daten werden im Stapelspeicher abgelegt, von wo sie entweder in das Unterprogramm oder in das Hauptprogramm übernommen werden. Diese Art der Datenübergabe wird häufig bei der Verwendung von Assemblerprogrammen zusammen mit einer höheren Programmiersprache (z.B. PASCAL) angewendet.

3. Hauptprogramm und Unterprogramm benutzen einen gemeinsamen Speicherbereich für die verwendeten Daten.

Welche der 3 Möglichkeiten verwendet wird, hängt von der vorgegebenen Programmieraufgabe ab. Wird ein Programm nur von einem Programmierer geschrieben, so wird er die Methode wählen, die ihm am besten geeignet erscheint. Arbeiten mehrere Programmierer an der gleichen Programmieraufgabe, so muß die Art der Datenübergabe vorher festgelegt werden.

Die Verwendung von Unterprogrammen bringt folgende Vorteile:
Ein längeres Programm wird in kleinere Programmstücke aufgeteilt. Diese sind leichter überschaubar und auch einfacher zu testen. Mit Unterprogrammen kann eine Programmbibliothek aufgebaut werden. Auf diese kann bei der Programmierung zurückgegriffen und somit Zeit gespart werden.

### 4.4 Indirekter Sprung und indirekter Unterprogrammsprung

In unserem Programm in Abbildung 4.2 ist in den Speicherzellen $FD18 – $FD1A mit 6C 38 00 ein indirekter Sprung JMP ($ 0038) programmiert. Indirekte Adressierung bedeutet, daß nicht auf die angegebene Adresse ($ 0038), sondern auf eine Adresse, die dort gespeichert ist, gesprungen wird. Dabei enthält $0038 des LSB und die folgende Speicherzelle ($ 0039) das MSB des Sprungziels.
Ist der Inhalt von $0038 zum Beispiel $00 und der Inhalt von $ 0039 $ EF, so wird mit JMP ($0038) nach $EF00 gesprungen.

# Programmieren in Maschinencode, Teil 5

**Programmieren in Maschinencode, Teil 5**

**5.1 Die indizierte Adressierung**

In diesem Teil wollen wir uns mit der indizierten Adressierung beschäftigen. Betrachten wir dazu folgendes Beispiel: Wir haben Daten (Zahlen, Buchstaben) in dem Speicherbereich $1000 bis $101F gespeichert. Diese Daten sollen nun in einen anderen Bereich, beginnend bei der Adresse $2000 geschrieben werden. Dafür könnte man folgendes Programm schreiben:

```
LDA  $1000     ; Lade den Inhalt von
STA  $2000     ; $1000, Speichere ihn
LDA  $1001     ; nach $2000
STA  $2001
LDA  $1002
STA  $2002
.
.
usw. bis
LDA  $101F
STA  $201F
```

Abgesehen von der Schreibarbeit für dieses Programm, so benötigt man in diesem Fall 6 Programmbyte für das Umsetzen eines Datenbytes von einer Adresse zu einer anderen. Für unsere 32 Datenbyte benötigt man also ein Programm von insgesamt 32 x 6 = 192 Byte. Sollen noch mehr Daten umgesetzt werden, so würde ein Programm in dieser Form sehr lang werden. Um die Programmierung nun zu vereinfachen, benötigt man für solche Programme zur Datenumsetzung die indizierte Adressierung mit einem der beiden Indexregister der 6502 CPU. Der Befehl LDA $1000,X bedeutet Hole den Inhalt der Speicherzelle, deren Adresse sich aus der Summe von Adresse ($1000) und Inhalt des X-Registers ergibt. Mit LDA $1000,X wird mit X = 1 der Inhalt der Speicherzelle $1001, mit X = 2 der Inhalt der Speicherzelle $1002 und mit X = $1C der Inhalt der Speicherzelle $101C geholt. Anstelle des X-Registers hätte man auch das Y-Register verwenden können. Der Befehl lautet dann LDA $1000,Y.
Für unser Beispiel erhalten wir dann folgendes Programm. (Abb. 5.1)

```
0800            3  ;
0800            4  ;*****************************
0800            5  ;*                           *
0800            6  ;*     VERSCHIEBEN           *
0800            7  ;*                           *
0800            8  ;*****************************
0800            9  ;
0800           10  ;
0800 A200      11        LDX #$00
0802 BD0010    12  M     LDA $1000,X         ;($1000) --> A
0805 9D0020    13        STA $2000,X         ;(A) --> $2000
0808 E8        14        INX
0809 E020      15        CPX #$20            ; (X)= $20 ?
080B D0F5      16        BNE M               ; NEIN,WEITER
080D 00        17        BRK
080E           18  ;
               19        END

***** END OF ASSEMBLY
```

Abb. 5.1

Das X-Register wird Null gesetzt und dann mit LDA $1000,X der Inhalt von $1000 geholt und nach $2000 geschrieben. Mit INX wird der Inhalt des X-Registers um Eins erhöht. Danach muß die Abfrage erfolgen, ob alle Daten übertragen worden sind. Insgesamt sollen die Inhalte der Zellen $1000 bis $101F übergeben werden. Die erste Zelle, die nicht übertragen werden soll, ist die Zelle $1020. Ist der Inhalt des X-Registers nach dem Erhöhen um Eins gleich $20, so wird das Programm abgebrochen.

Ein Wort noch zu dem Kommentar in diesem Beispiel. Mit $1000 wird die Adresse einer Zelle eingegeben, mit ($1000) der Inhalt dieser Zelle.
Die beiden Indexregister X und Y sind 8-Bit-Register, somit ist eine Indizierung von 0 bis 256 möglich. Datenfelder mit maximal 256 Byte können mit dieser Adressierung übertragen werden. Sollen größere Datenfelder umgespeichert werden, so müssen andere Adressierungsarten, auf die wir gleich eingehen werden, verwendet werden.

Zuvor noch ein Programmbeispiel, bei dem die Inhalte der Zellen $1000 bis $10FF vertauscht werden. Also der Inhalt von $1000 mit $10FF, von $1001 mit $10FE, von $1002 mit $10FD usw.
(Abb. 5.2)

Zuerst wird 0 nach X und FF nach Y geschrieben. Der Inhalt von $100 wird geholt und im Stapelspeicher zwischengespeichert. Dann wird der Inhalt von $10FF nach $100 umgespeichert und der zwischenge-

speicherte Wert nach $ 10FF geschrieben. Das Y-Register wird um Eins erniedrigt, das X-Register um Eins erhöht. Die Vertauschung ist abgeschlossen, wenn der Inhalt von X = $80 ist.

```
0800            3  ;
0800            4  ;*****************************
0800            5  ;*                           *
0800            6  ;*     VERTAUSCHEN           *
0800            7  ;*                           *
0800            8  ;*****************************
0800            9  ;
0800           10  ;
0800 A200      11         LDX #$00
0802 A0FF      12         LDY #$FF            ;FF --> YREG
0804 BD0010    13  M      LDA $1000,X         ;($1000+X) --> A
0807 48        14         PHA                 ;(A) --> STAPEL
0808 B90010    15         LDA $1000,Y         ;($1000+Y) --> A
080B 9D0010    16         STA $1000,X         ; (A) --> $1000+X
080E 68        17         PLA                 ;(STAPEL) --> A
080F 990010    18         STA $1000,Y         ;(A) --> $1000+Y
0812 88        19         DEY                 ;(Y)-1 --> Y
0813 E8        20         INX                 ;(X)+1 --> X
0814 E080      21         CPX #$80            ; FERTIG ?
0816 D0EC      22         BNE M
0818 00        23         BRK
0819           24  ;
               25         END

***** END OF ASSEMBLY
```

**Abb. 5.2**

Bei der indizierten Adressierung wird zur Berechnung der endgültigen (effektiven) Adresse die Summe aus programmierter Adresse und Inhalt des Indexregisters gebildet. Tritt bei dieser Summenbildung ein Übertrag auf, so wird dieser berücksichtigt. Mit X = $FF wird durch den Befehl
STA $ 10E0,X der Akkumulatorinhalt nach $11DF geschrieben.
Der Befehlssatz der 6502 CPU besitzt noch zwei weitere Adressierungsarten, die sich aus der indirekten und der indizierten Adressierung zusammensetzen. Zur Erinnerung nochmal: Bei der indirekten Adressierung ist nicht die programmierte Adresse, sondern der Inhalt dieser Adresse die eigentliche Zieladresse.

Beispiel: JMP ($ 2000) bedeutet einen Sprung nach $ 3AEF, wenn dies der Inhalt der Zelle $2000 ist.

### 5.2 Die indiziert-indirekte Adressierung

Bei dieser Adressierung ist die programmierte Adresse immer eine Adresse aus der Zero-Page und das verwendete Register ist immer das X-Register.

Beispiel:LDA ($10,X)

Die endgültige Adresse wird nun auf folgende Weise berechnet:

Zur programmierten Adresse $ 10 wird der Inhalt des X-Registers hinzuaddiert. Der Inhalt dieser neuen Adresse und des darauffolgenden Bytes ist die endgültige Adresse. Betrachten wir dazu folgendes Beispiel:

Der Inhalt der Zellen $0E – $15 ist:

(0E) = FF
(0F) = 0F
(10) = 00
(11) = 11
(12) = 2F
(13) = 30
(14) = 00
(15) = 47

Der Befehl LDA ($ 10,X) holt mit X = 0 den Inhalt der Zelle $ 1100, mit X = 2 den Inhalt der Zelle $ 302F und mit X=4 den Inhalt der Zelle $4700.

Tritt bei der Berechnung der Summe aus Adresse und Registerinhalt ein Übertrag auf, so wird dieser **nicht** berücksichtigt.
Deshalb wird mit X = $ FE der Inhalt von $0FFF geholt.

### 5.3 Die indirekt-indizierte Adressierung

Auch hier ist die programmierte Adresse eine Adresse aus der Zero-Page. Als Indexregister kann nur das Y-Register verwendet werden.
Beispiel: STA ($20), Y

Die endgültige Adresse wird wie folgt berechnet: Zum Inhalt der programmierten Adresse in den Zellen $ 10, $ 11 wird der Inhalt des Y-Registers addiert. Dies ist dann die gültige Adresse.
Mit ($20) = 3E
($21) = 2F

und X = 0 wird der Inhalt des Akkumulators nach $ 2F3E, mit X = $ 10 der Inhalt nach $2F3F gespeichert.

Beide Adressierungsarten werden in dieser vollständigen Form nicht sehr häufig in Programmen angewendet. Allerdings trifft man sie in der Form der indirekten Adressierung an. Setzt man X oder Y-Register Null, so bedeutet LDA ($ 10,X): Hole den Inhalt der Zelle, deren Adresse in $ 10 und $ 11 gespeichert ist. Das gleiche bedeutet auch der Befehl LDA ($ 10),Y mit Y = 0.

Ändert man den Inhalt dieser Zellen, so kann man mit einem Befehl mit einer fest programmierten Adresse auf verschiedene Adressen zugreifen. Die soll nun in einem kleinen Programm verwendet werden, das nicht nur 256 Byte, sondern 4K Byte von $1000 nach $2000 verschiebt. (Abb. 5.3)

```
0800            3  ;
0800            4  ;****************************
0800            5  ;*                          *
0800            6  ;* 4K VERSCHIEBEN           *
0800            7  ;*                          *
0800            8  ;****************************
0800            9  ;
0800           10  ;
0800 A200      11        LDX #$00          ;0 --> X
0802 8610      12        STX $10           ;(X) --> LO BYTE START
0804 8612      13        STX $12           ;(X) --> LO BYTE ZIEL
0806 A910      14        LDA #$10          ;$10 --> A
0808 8511      15        STA $11           ;(A) --> HI BYTE START
080A A920      16        LDA #$20
080C 8513      17        STA $13           ;(A) --> HI BYTE ZIEL
080E           18  ;
080E A110      19  M     LDA ($10,X)       ;(($10)) --> A
0810 8112      20        STA ($12,X)       ;(A) --> ($12)
0812 E610      21        INC $10           ;($10)+1 --> $10
0814 E612      22        INC $12           ;($12)+1 --> $12
0816 D0F6      23        BNE M             ; WENN <> 0 ,WEITER
0818 E611      24        INC $11           ; SONST ($11)+1 --> $11
081A E613      25        INC $13           ; ($13)+1 --> $13
081C A511      26        LDA $11
081E C920      27        CMP #$20
0820 D0EC      28  *     BNE M
0822 00        29        BRK
0823           30  ;
               31        END

***** END OF ASSEMBLY
```

**Abb. 5.3**

Im Programm werden zuerst die Anfangsadressen für Start ($ 10, $ 11) und Ziel ($ 12, $ 13) festgelegt. Dann wird mit LDA ($ 10,X) der Inhalt von $ 1000 geholt und mit STA ($ 12,X) nach $ 2000 geschrieben. Der Zelleninhalt von $ 10 und $ 12 wird um Eins erhöht. Wenn dabei nach der Übertragung von 256 eine Page Grenze überschritten wird, muß auch der Zelleninhalt von $ 11 und $ 13 erhöht werden, bis in den Zellen $ 10 und $ 11 die Adresse der ersten, nicht zu verschiebenden Zelle ($ 2000) erreicht ist.

Zur Programmierübung 2 kleine Aufgaben:

1.) Programm FILL. Ein Speicherbereich, mit der Anfangsadresse in $ 10 und $ 11 und der Endadresse in $ 12 und $ 13 soll mit einer Hexzahl, die in $ 14 steht, gefüllt werden.

2.) Programm MOVE. Ein Datensatz mit der Anfangsadresse in $ 10, $ 11 und der Endadresse in $ 12, $ 13 soll zu einer Zieladresse gespeichert und in $ 14, $ 15 verschoben werden. Für diese Zieladresse gibt es keine Beschränkungen. Sie darf auch innerhalb des Adressbereichs des Datensatzes liegen. Dies ist bei den obigen Beispielen nicht möglich.

**Stichpunkte zu Teil 5**

- Indizierte Adressierung
- Indiziert-indirekte Adressierung
- Indirekt-indizierte Adressierung
- Verschieben von Daten im Speicher

**Ekkehard Flögel** ■